| | |
|---|---|
| **Titre:** Title: | Balancing Software Maintainability and Performance: An Empirical Study on Refactoring Practices |
| **Auteur:** Author: | Nana Gyambrah |
| **Date:** | 2025 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Gyambrah, N. (2025). Balancing Software Maintainability and Performance: An Empirical Study on Refactoring Practices [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/65495/ |

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

# Balancing Software Maintainability and Performance: An Empirical Study on Refactoring Practices

**NANA GYAMBRAH**
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Mai 2025

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

Ce mémoire intitulé :

**Balancing Software Maintainability and Performance: An Empirical Study on Refactoring Practices**

présenté par **Nana GYAMBRAH**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

**Maxime LAMOTHE**, membre et directeur de recherche
**Heng LI**, président
**Mohammad HAMDAQA**, membre

# DEDICATION

*To my beloved father,*
*whose light continues to illuminate my path*
*even in his absence.*

*To my mother and uncle,*
*Diana and Kwadwo,*
*for your constant support and prayers...*

## ACKNOWLEDGEMENTS

I am profoundly grateful to my research supervisor, Professor Maxime Lamothe, whose expert guidance, and patient mentorship have been instrumental in shaping this thesis. Your dedication to academic excellence and willingness to share your knowledge have been truly inspiring.

I would like to express my deepest gratitude to my mother, whose unwavering support, endless encouragement, and countless sacrifices have been the foundation of all my achievements. Her strength and resilience continue to inspire me every day.

I dedicate this work to the loving memory of my father, who, though no longer with us, planted the seeds of curiosity and perseverance in me from an early age. His belief in the power of education and his dreams for my future have been a guiding light throughout this journey. To my uncle, who stepped into a paternal role and provided invaluable guidance and support when I needed it most – your generosity and wisdom have meant more than words can express.

To the members of my thesis jury, I extend my sincere appreciation for your time, expertise, and thoughtful evaluation of my work.

This achievement would not have been possible without each of you.

# RÉSUMÉ

Le développement logiciel moderne est souvent confronté à des défis de performance découlant d'un code sous-optimal, d'algorithmes inefficaces ou d'une gestion inadéquate des ressources. Le refactoring, technique traditionnellement employée pour améliorer la qualité et la maintenabilité du code sans en modifier le comportement, peut contribuer à atténuer ces problèmes. Alors que la recherche s'est largement penchée sur le lien entre refactoring et bugs fonctionnels, son influence sur la performance reste relativement peu étudiée.

Dans ce mémoire, nous présentons deux études empiriques complémentaires examinant la relation entre les pratiques de refactoring et la performance logicielle. La première étude porte sur 255 projets Java open source, analysés à partir de données issues de l'API de GitHub et de RefactoringMiner. Nos résultats indiquent que les problèmes de performance sont fréquemment résolus par des modifications de la structure des classes et de la configuration des méthodes, avec l'opération « Change Variable Type » comme stratégie de refactoring la plus marquante. L'analyse statistique révèle que les commits liés à la performance sont 1,77 fois plus susceptibles de contenir des opérations de refactoring que les commits non liés à la performance. De plus, une classification manuelle de 300 instances de refactoring ciblant la performance met en évidence 11 motivations distinctes pour le refactoring dans ces contextes, ainsi que trois environnements principaux de mise en œuvre.

Fortes de ces constatations, nos recherches se poursuivent dans une deuxième étude consacrée à l'impact du refactoring sur le temps d'exécution dans 15 projets Java open source. À l'aide d'outils automatisés pour tracer les modifications de code et mesurer la performance, nous identifions les types de refactoring les plus souvent associés à des fluctuations de performance en particulier l'extraction de méthodes et l'inlining de variables et évaluons l'ampleur de ces changements. Nous soulignons également des facteurs contextuels qui influent sur la direction et l'ampleur des effets de performance, tels que la structure du code, la fréquence d'invocation des méthodes et la couverture de tests.

Globalement, nos résultats soulignent la nature nuancée et ambivalente du refactoring : bien qu'il améliore la maintenabilité, il peut aussi avoir un impact significatif sur les performances, engendrant parfois des gains notables et occasionnellement des régressions. En proposant une méthodologie reproductible pour évaluer les variations de performance liées au refactoring, cette recherche offre des orientations précieuses aux développeurs et aux chercheurs désireux de concilier la qualité du code et l'efficacité à l'exécution.

# ABSTRACT

Modern software development often confronts performance challenges stemming from suboptimal code, inefficient algorithms, or inappropriate resource management. Software refactoring, a technique traditionally employed to enhance code quality and maintainability without altering software behavior, has the potential to mitigate these issues. While prior research has extensively investigated the relationship between refactoring and functional defects, its influence on performance remains comparatively underexplored.

In this thesis, we present two complementary empirical studies that examine how refactoring practices relate to and affect software performance. The first study investigates 255 Java open-source projects, drawing on data from GitHub's API and RefactoringMiner. Our findings indicate that performance-related issues are frequently addressed through modifications to class structures and method configurations, with "Change Variable Type" emerging as the most impactful refactoring strategy. Statistical analysis reveals that performance-related commits are 1.77 times more likely to contain refactoring operations than non-performance commits. A deeper manual classification of 300 performance-refactoring instances uncovers 11 distinct motivations for refactoring when tackling performance concerns, as well as three main contexts in which these refactorings are applied.

Building on these insights, the second study focuses on refactoring's effect on execution time in 15 open-source Java projects. By employing automated tools to trace code changes and measure performance, we identify the most common refactoring types associated with performance fluctuations—particularly method extraction and variable inlining—and assess the magnitude of these changes. We further highlight contextual factors that shape the direction and degree of performance impacts, including code structure, method invocation frequency, and test coverage.

Collectively, our results underscore the nuanced, dual impact of refactoring: while it enhances maintainability, it can also significantly influence software performance, sometimes leading to notable gains and occasionally to regressions. By providing a replicable methodology for evaluating refactoring-induced performance variations, this work aims to offer valuable guidelines for developers and researchers seeking to strike an optimal balance between code quality and runtime efficiency.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS AND ACRONYMS

| | |
|---|---|
| API | Application Programming Interface |
| JVM | Java Virtual Machine |
| JMH | Java Microbenchmark Harness |
| MCR | Modern Code Review |
| PR | Pull Request |
| SLR | Systematic Literature Review |
| JPerfEvo | Java Performance Evolution |
| ANN | Artificial Neural Network |
| SRM | Source Role Model |
| TRM | Target Role Model |
| AI | Artificial Intelligence |
| XML | Extensible Markup Language |
| AR | Association Rule Mining |

# LIST OF APPENDICES

# CHAPTER 1 INTRODUCTION

In the ever-evolving landscape of software development, the quest for high-performing, efficient, and reliable systems remains a paramount concern for developers and end-users alike [1, 2]. Performance issues, considered a subclass of software defects that adversely affect runtime behavior and responsiveness, pose significant challenges in meeting user expectations [1, 2, 3]. These issues frequently stem from suboptimal code implementations, inefficient algorithms, or inadequate resource management, resulting in prolonged execution times, increased resource consumption, and degraded user experiences [2]. Consequently, researchers and practitioners have long sought effective strategies to mitigate performance-related concerns and optimize software performance [1, 4, 5].

Maintaining high-quality, understandable, and flexible codebases as software systems grow and evolve becomes increasingly important [6, 7]. Refactoring has emerged as a fundamental practice in software engineering, aiming to improve code maintainability, readability, and sustainability without altering the functional behavior of the system [8, 9, 10, 11, 12, 13, 14]. By transforming code structure and eliminating technical debt, refactoring can reduce the likelihood of functional bugs [15, 16, 17, 18, 19]. Yet, recent studies have shown that refactoring may significantly impact software performance either positively or negatively, depending on the type of changes made and the specific context [5, 20, 21, 22]. While some refactorings inadvertently alter low-level runtime execution details, memory usage, or compiler optimizations [23], others may improve performance by streamlining code paths.

Despite these insights, a systematic and comprehensive understanding of how different refactoring operations correlate with performance outcomes remains limited. On one hand, Traini et al. [20] have analyzed refactored code components exercising performance benchmarks, illustrating the trade-offs that certain refactoring operations can introduce. However, more empirical evidence is needed to ascertain *when* and *why* performance changes occur, as well as how frequently refactoring is employed to address performance-related issues in practice. Indeed, existing research has traditionally emphasized performance bug detection and fixes [24, 25], rather than focusing on the developer perspective and the specific refactoring choices made to resolve performance concerns.

To address these gaps, this thesis presents two complementary empirical investigations into the relationship between refactoring and software performance. First, we conduct an exploratory study involving 255 Java open-source projects, gathering performance-related issues, commits, and pull requests from GitHub's API. Leveraging RefactoringMiner [26, 27],

we mine our subject systems for both performance and non-performance refactorings, ultimately building a comprehensive dataset for statistical analysis. We quantify how often refactoring is used to address performance issues, identify the most prevalent types of refactorings in performance-related contexts, and uncover the motivations that lead developers to refactor in these scenarios. We further employ *Association Rule Mining*—specifically the *Apriori algorithm* [28]—to discover common patterns of refactoring types in both performance and general refactorings.

Second, we perform a systematic investigation of refactoring's impact on execution time within 15 Java open-source projects. Using a fully automated pipeline built on RefactoringMiner [26, 27] and the JPerfEvo[1] tool, we track code changes, filter commits, and measure performance variations under controlled conditions. In total, we analyze 739 commits, capturing data on 1,559 method-level changes and 11,317 refactorings across 90 unique refactoring types. By leveraging the Java Microbenchmark Harness (JMH) module within each project and a custom Java instrumentation agent, we obtain fine-grained insights into how refactoring influences minimum, maximum, and average execution times in real-world scenarios. Our findings reveal that method extraction and variable inlining are most frequently associated with performance changes—sometimes yielding notable gains, other times introducing regressions, particularly in complex code paths. We also manually investigate a subset of these performance data points to identify contextual factors, such as code structure and method invocation frequency, that shape refactoring outcomes.

Collectively, these two studies provide a multifaceted view of refactoring's role in performance optimization. By examining both the prevalence of refactoring in performance-related issues and the direct impact on execution times, we offer new insights and actionable guidance for developers aiming to balance maintainability with runtime efficiency. Ultimately, this thesis seeks to bridge the gap between the practice of refactoring and the evolving demands of high-performance software systems, informing best practices and strategies for effectively addressing performance issues through targeted code transformations.

### 1.1 Research Objectives

This thesis, including Chapters 3 & 4, aims to tackle the following research objectives:

- Measure the prevalence of refactoring in performance-related commits
- Identify how and why refactoring is used in performance-related commits

[1] https://github.com/kavehshahedi/java-performance-history-buddy

- Determine which refactoring types coincide with performance changes
- Measure the effects of refactoring on performance changes

## 1.2 Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 2**
  This chapter presents a literature review of empirical studies on refactoring, performance bug detection, and the relationship between refactoring and software quality. It also discusses recent advances in performance-oriented refactoring and relevant detection and testing tools.
- **Chapter 3**
  This chapter explores the relationship between refactoring and performance-related issues in software systems through an empirical study measuring how and why developers implement refactoring in performance-related commits. The study quantifies refactoring types usage specifically within performance-related issues.
- **Chapter 4**
  This chapter investigates the impact of refactoring on software performance, specifically execution, using a robust data collection tool. The study provides insights into the effects and implications of refactoring on performance change.
- **Chapter 5**
  This chapter discusses the threats to validity that may impact the reliability and generalizability of the studies' findings, addressing potential limitations in the data, methodology, and analysis.
- **Chapter 6**
  This chapter concludes the thesis by summarizing the key findings, highlighting the studies' contributions and implications, and outlining their limitations and directions for future work.

# CHAPTER 2 LITERATURE REVIEW

## 2.1 Empirical Studies on Refactoring

Refactoring has been a central topic in software engineering for decades, attracting considerable attention due to its potential to improve code quality, maintainability, and extensibility. Early work by Opdyke [14] laid the groundwork for refactoring in object-oriented frameworks, and Fowler [6] provided a comprehensive catalog of refactoring types, sparking a surge in empirical studies on the subject. For example, Coelho et al. [29] examined refactoring-inducing pull requests (PRs) in Modern Code Review, revealing that such PRs differ substantially from non-refactoring-inducing ones in multiple aspects, thus suggesting opportunities for improving pull-based code review practices. In a study of internal quality attributes, Chávez et al. [30] found that 65% of refactoring operations improved metrics like cohesion and complexity, while 35% left them unchanged.

Other researchers have focused on the motivation and tools behind refactoring. Pantiuchina et al. [31] proposed a taxonomy of reasons developers choose to refactor, whereas Murphy-Hill et al. [32] studied the refactoring tools employed by developers to gain insights into common workflows. Large-scale reviews have further synthesized empirical research in this area: [33] conducted a broad systematic literature review of over 3,000 papers, capturing historical trends and the expansion of refactoring research into domains such as cloud, mobile, and web applications. Similarly, Akhtar et al. [34] explored refactoring techniques, challenges, and best practices, underscoring the need for additional empirical validation. A tertiary review by Alotaibi and Mkaouer [35] combined findings from multiple secondary studies, highlighting persistent gaps such as the limited exploration of refactoring's direct impact on software performance.

In parallel, studies employing search-based and automation-focused approaches have also emerged. Mohan and Greer [36] analyzed search-based refactoring optimization, while Baqais and Alshayeb [37] surveyed automated refactoring techniques and tools. Lui [38] emphasized the importance of preserving code reliability during refactoring, drawing parallels to compiler design principles. Peruma et al. [39] and Almogahed et al. [40] investigated real-world developer challenges in selecting the most suitable refactoring strategies, revealing performance trade-offs that often complicate decision-making. Further diversifying the field, Motogna et al. [41] and Nyirongo et al. [42] explored how artificial intelligence and deep learning techniques could automate more complex refactoring tasks. Finally, Golubev et al. [43] surveyed industry practitioners to understand tool adoption and practical hurdles, whereas AlOmar

et al. [44] examined how standardized refactoring strategies improve code reusability.

By covering a broad spectrum from theoretical principles to empirical observations, this body of work highlights the multifaceted role of refactoring in improving various quality attributes. However, its direct implications for performance remain less thoroughly examined, a gap that more recent investigations are starting to address.

## 2.2 Detection and Fixing of Performance Bugs

Performance issues constitute a critical subclass of software defects, influencing user experience, resource usage, and overall runtime behavior [1, 2, 3]. Considerable effort has been invested in studying how these bugs are detected and resolved [1, 45, 46, 47, 48, 49]. Various tools and approaches have been introduced to enhance performance bug detection, yet Linares-Vásquez et al [24] observed that while developers often rely on performance bottleneck profilers, these tools may not effectively isolate and address specific performance bugs because they focus primarily on general profiling rather than targeted bug identification. Zaman et al. [25] conducted a qualitative examination of 400 performance and non-performance bugs, noting that performance-related defects often take more time to replicate, discuss, and fix. Interestingly, developers at times accept performance regressions to achieve other improvements in the codebase.

Studies have offered different perspectives on how quickly performance bugs are resolved relative to other bug types. For instance, some prior works found evidence that performance bugs can, in some contexts, be addressed more rapidly than security bugs [50, 51, 52]. Kumar et al. [53] examined how bug fixing affects internal quality attributes, reporting that more than 80% of classes with at least one "critical" attribute also contained bugs. By highlighting these distinctions, the literature underscores the unique challenges in identifying and fixing performance bugs, as opposed to non-performance defects. Our own work contributes to this conversation by exploring how refactoring operations specifically support or complicate the process of resolving performance-related issues.

## 2.3 Refactoring and Software Quality

Prior investigations have addressed the broader question of how refactoring activities impact software quality, revealing both positive and negative effects. Traini et al. [20] performed an empirical study of more than 20 Java open-source projects, discovering that certain refactorings like Extract Class and Extract Method could reduce performance in performance-critical contexts. They also observed that methods were rarely refactored, regardless of whether they

were deemed performance-relevant or not. In contrast, Chávez et al. [30] highlighted scenarios in which the majority of refactorings led to improvements in internal quality attributes such as cohesion and complexity. Meanwhile, work on related non-functional aspects, like energy consumption and data access, further affirms that code transformations can have unexpected repercussions. Anwar et al. [54] and Ournani et al. [55] found that refactorings sometimes lower energy consumption, but the impact on performance was inconclusive. Sahin et al. [56] demonstrated that even commonly used refactorings can influence energy usage, with Extract Local Variable providing consistent benefits.

Several researchers have turned their attention to how refactoring specifically targets performance or performance antipatterns. Iannone et al. [57] introduced an automated system that detects and refactors problematic code snippets linked to performance bottlenecks. Arcelli et al. [5] employed a model-based perspective, defining Source Role Models (SRMs) for performance antipatterns and Target Role Models (TRMs) for optimized replacements, validated through an e-commerce case study. Muse et al. [58] zeroed in on data access refactoring, emphasizing code quality improvements without necessarily addressing the underlying data access inefficiencies. Kannangara and Wijayanayake [59] investigated ten refactoring types, linking "Replace Conditional with Polymorphism" to improved external code quality while cautioning that "Introduce Null Object" could undermine performance or resource utilization.

### 2.4 Recent Advances in Performance-Centric Refactoring

More recent empirical works have begun to examine how specific refactoring operations correlate with performance changes in diverse contexts. Agnihotri and Chug [21] reported that "batch refactoring" efforts, especially at the method level, could yield a sizable quality boost, suggesting that developers frequently refactor code sections already identified as problematic. Siegmund et al. [60] proposed a machine learning-based approach to model performance fluctuations as configuration options change, underscoring the intricate interplay between refactoring decisions and runtime characteristics. Imran et al. [22] [61] adopted a resource-aware lens, employing ANN-based models to predict the CPU and memory consequences of refactoring "code smells" in open-source applications. Taken together, these studies demonstrate a growing recognition that careful refactoring can function as a potent lever for performance optimization, even if the potential risks of performance regressions or other side effects cannot be ignored.

### 2.5 Software Performance Detection, Testing, and Tooling

Beyond the act of refactoring itself, a rich body of work addresses how performance issues can be identified and validated in software systems. Techniques and best practices for performance detection and testing are well-documented in studies [62, 63, 64, 65, 66], with a focus on Java applications and broader distributed systems. Tools for in-depth performance analysis have also proliferated. For example, Li et al. [67] introduced DJXPerf, a lightweight profiler that pairs hardware monitoring counters with Java object-level insights at minimal overhead (about 8.5% runtime and 6% memory). A similar profiler, JXPerf, relies on hardware monitoring units and debug registers, attributing performance costs to both the machine code and the corresponding Java source code.

Researchers have proposed methodologies for identifying performance antipatterns and converting raw metrics into actionable guidance. Cortellessa et al. [68] automated the generation of performance feedback by encoding antipatterns as logical predicates over XML system descriptions, moving beyond standard statistical analyses. Avritzer et al. [69] advocated workload characterization to inform performance testing strategies, and Munson and Khoshgoftaar [70] examined how fault-prone modules contribute to runtime inefficiencies.

Performance benchmarking and warm-up detection have also garnered interest. Leitner and Bezemer [71] explored common challenges in performance testing across open-source Java projects, such as inconsistent testing environments. Traini et al. [72] employed time series classification to dynamically detect when Java microbenchmarks reach a steady state, improving upon static warm-up estimates in up to 35.3% of tested benchmarks. A subsequent study showed that many benchmarks fail to attain a genuine steady state, adding a layer of complexity to benchmarking best practices [73]. Complementary efforts examined how tools like Apache JMeter and unit test frameworks could be extended or adapted to capture performance nuances more effectively [74, 75].

Collectively, these developments underscore the importance of reliable tooling and rigorous methodologies for performance detection and verification. When integrated with insights on how certain refactoring operations affect execution times, developers can more systematically identify, implement, and validate performance-centric refactorings.

# CHAPTER 3 AN EMPIRICAL STUDY OF REFACTORINGS RELATED TO PERFORMANCE ISSUES

## 3.1 Introduction

Software performance issues, arising from suboptimal code, inefficient algorithms, or improper resource management, can significantly degrade system efficiency and user experience [1, 2, 3]. While refactoring is widely adopted to improve code maintainability and quality [8, 9, 11], its role in addressing performance-related issues remains underexplored. Although prior studies have examined refactoring's impact on software quality attributes such as cohesion and complexity [30, 31], limited research systematically investigates its prevalence, motivations, and patterns in performance-critical contexts.

This study seeks to bridge this gap by analyzing the relationship between refactoring and performance-related issues in software systems. We conduct an empirical investigation of 255 open-source Java projects, utilizing GitHub's API and RefactoringMiner to extract and analyze performance-related commits, pull requests, and refactoring operations. Our statistical analysis reveals that refactoring is 1.77 times more likely to occur in performance-related commits than in non-performance commits, with class restructuring and method modifications, particularly "Change Variable Type", "Add Parameter", and"Move Class" being the most frequently applied refactoring types. Through a manual classification of 300 performance-related refactorings, we identify 11 motivations and three primary contexts in which developers employ refactoring to address performance issues.

To guide our investigation, we formulate the following research questions:

- **RQ1:** What is the likelihood of finding refactoring occurrences in performance-related commits compared to non-performance-related commits?
- **RQ2:** What refactoring types are most commonly used when addressing performance issues?
- **RQ3:** Do performance issues affect the choice of refactoring types used by developers?
- **RQ4:** What are the patterns of co-applied refactoring types in performance-related and non-performance-related refactorings?
- **RQ5:** What are the different reasons for refactoring in performance issues?

By addressing these questions, this study provides empirical insights into how developers leverage refactoring to enhance software performance. Our findings contribute to both research and practice by informing best practices for performance-aware refactoring strategies and guiding future tool development to support performance-driven software evolution.

### 3.2 Study Design

This section describes the design of our study. To answer our RQs, we focus on the unique aspects of performance refactorings. Here, we discuss the procedures involved in our project selection and data collection. An overview of our data collection process can be found in Figure 3.1. The results of each of the steps in our process can be found in our online replication package.[1]

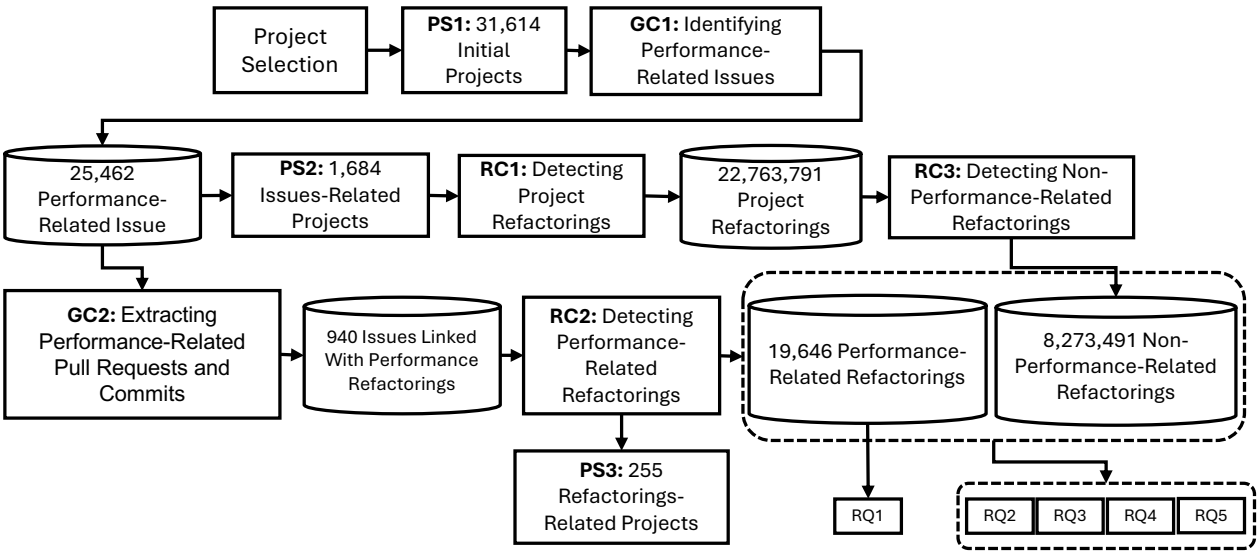


Figure 3.1 Overview of our refactoring collection process

#### Projects Selection

We conducted our study on open-source Java projects hosted on GitHub. However, there are more than 12 million GitHub repositories primarily written in Java, due to resource constraints we could not leverage the entire population. Our goal was therefore to sample active projects with regular maintenance activities, and public engagement. Using the GitHub API, we focused on projects with more than 20 stars (indicating popularity) and that were not archived (indicating active maintenance or development), following the approach of prior works [76]. We excluded forked repositories since they are copies of original repositories and

[1] `https://github.com/quamnana/refactoring_related_to_performance_issues`

are mostly used to experiment with changes without affecting the originals [76]. We initially collected 31,614 projects and filtered out repositories smaller than 5MB. This filtering was done because larger projects are more likely to contain substantial, meaningful codebases. Projects under 5MB might be too small to provide significant insights or represent fully developed software. Our focus on larger repositories ensures a more robust analysis of established and comprehensive projects. Hence we finally collected 11131 projects through this process. Further in our data collection, we classified these selected projects into three categories:

1. **PS1: Initial Projects:** are projects that were initially collected from GitHub.
2. **PS2: Issues-Related Projects:** are projects with performance-related issues, extracted from initial collected projects in **GC1**.
3. **PS3: Refactorings-Related Projects:** are projects with performance-related refactorings within their repository, extracted from Issues-Related Projects in **RC2**.

An overview of these projects is shown in Table 3.1. The column size indicates the size of the whole repository (including all of its history), in kilobytes. The columns #open issues, #closed issues, and #stars show the numbers of open issues, closed issues, and stars of the projects respectively. The details of the categorized projects are also shown in Table 3.2

#### Gathering Performance-Related Issues, Pull Requests and Commits

Issues and pull requests are both features commonly found in online version control platforms like GitHub, GitLab, and Bitbucket.

1. **Issues:** are elements within a version control system used to track tasks, bugs, issues, ideas, or enhancements related to a project. They serve as a means for discussion and management of work.

Table 3.1 Statistical overview of Initial Projects collected using the GitHub API

| | Size (KB) | #Open issues | #Closed issues | #Stars |
|---|---|---|---|---|
| min | 5,002 | 0 | 0 | 20 |
| 25% | 10,059 | 1 | 0 | 30 |
| median | 21,732 | 5 | 3 | 77 |
| 75% | 60,161.5 | 25 | 34 | 556 |
| max | 6,823,318 | 4,741 | 33,227 | 143,056 |

2. **Pull Requests:** are proposals to merge changes from one branch into another within a version control system. They facilitate collaborative development by allowing contributors to suggest modifications to a project's codebase.

1. **GC1 Identifying Performance-Related Issues:** To begin understanding how refactorings relate to performance issues, we first extract performance-related issues from our initial projects **PS1** as described in 3.2. Our goal is to find performance-related issues that are: (1) linked to a pull request (indicating that a proposed resolution was submitted for merging into the repository), and (2) closed (indicating that the issue was resolved and the corresponding code changes were merged into the repository). Using the GitHub API, we designed our search query to find performance-related issues of each project based on: (1) their full name; (2) performance-relevant keywords as used in prior works [1] [77] [78], including: *performance, fast, slow, perform, latency, throughput, optimize, speed, heuristic, waste, efficient, execution, too many times, lot of time, too much time, caching*; and (3) bug-related keywords as suggested in previous studies [79], which includes: *error, bug, issue, mistake, incorrect, fault, defect, flaw.* Thus, an issue is selected only if it matches at least one term from the performance-related keywords AND at least one term from the bug-related keywords, enabling us to narrow in on performance-related defects that were not explicitly labelled. To validate this keyword-based filter, we manually inspected a random sample of 200 retrieved issues across our projects. We found a precision of 88 % (i.e., 176/200 issues truly concerned performance faults) and a recall of 74 % (i.e., we recovered 74 % of all performance-related issues present in the sample projects' Github issues), yielding an $F_1$ score of 0.80. This approach enabled us to identify performance-related issues that were not explicitly tagged in their respective repositories. For instance, in the RobotBuilder project, there are 12 labels used to categorize issues, none of which specifically pertain to performance-related problems. These labels include: *bug, C++, duplicate, enhancement, help wanted, invalid, Java, New Commands, Old Commands, question, spam, and wontfix.*

   We initially identified 67,184 performance-related issues. For each of these issues, we extracted detailed information about both the issue and its project repository, including the *ID, name, number, URLs, title, and timestamps.* We excluded duplicate instances of performance-related issues resulting from the matching of specific search queries. Consequently, we retrieved 25,462 performance-related issues after omitting the duplicates. After identifying the performance-related issues, we analyzed them to determine which of our initial projects **PS1** had associated performance-related issues. We found

performance-related issues for only 1,684 of the initial projects **PS1**. We then classified these projects as a different dataset and labelled them as Issues-Related Projects **PS2** as described in 3.2. To understand why performance-related issues were not identified for some of the initial projects **PS1**, we randomly selected and manually inspected 50 of these projects. Our observations were as follows: (1) Some projects do not track their issues on GitHub but instead use other issue-tracking platforms like Jira and Bugzilla. For example, most Apache projects track their issues in Jira, making them inaccessible on GitHub, (2) Some projects had no reported performance issues from users, (3) Other projects had their issues reported in languages other than English, which likely made them hard to find for the GitHub API's search endpoint.

2. **GC2 Extracting Performance-Related Pull Requests and Commits:** Resolving software issues, whether performance-related or not, generally requires some change to a project's source code. This can be done through pull requests that can be verified and vetted by project maintainers before being merged into a project's repository. Thus, to identify candidate refactorings related to performance-related issues, we examine the pull requests that resolved the performance-related issues identified in **GC1**. The GitHub API offers an endpoint that provides detailed events for each issue, offering valuable insights into its resolution roadmap. By analyzing these events, we can gain a deeper understanding of each performance-related issue's evolution and identify the relevant pull request that resolved or aimed to resolve it. We identified a total of 22,527 pull requests from the 25,462 performance-related issues. For each pull request, we extracted detailed information about the pull request; *the pull request number, URLs, title, and timestamps* and updated this information to the pull request's corresponding performance-related issue identified earlier in **GC1**.

   In addition, we used the *pull request number* to extract detailed information about the commits associated with each pull request. Specifically, for each pull request, we identified and extracted the commit IDs of all related commits. This process helps

Table 3.2 Details of the categorized of projects

| **Metric** | **Description** | **Count** |
|---|---|---|
| Initial Projects | Total number of projects that were initially collected from GitHub. | 11,131 |
| Issue-Related Projects | Number of projects with performance-related issues. | 1,684 |
| Refactorings-Related Projects | Number of projects with performance-related refactorings within their repository. | 255 |

us provide a clear link between pull requests and their corresponding performance refactoring (we extract these in **RC2**).

### Refactoring Collection

To collect performance-related refactorings we rely on RefactoringMiner version 3.0.4 [26, 27] which can detect 100 refactoring types with a total Precision and Recall of 99.8% and 98.1% respectively. Given a GitHub URL and commit ID or pull request number of a Java project, RefactoringMiner reports the commits featuring refactorings, the refactoring types, and the files and lines affected by the refactoring. We collect refactorings in three distinct scenarios.

1. **RC1 Detecting Project Refactorings:** The first step to identify both performance and non-performance-related refactorings that are available in a project's repository is to extract Project Refactorings. **Project Refactorings** are refactorings that have taken place over the cause of a projects's commit history, these refactorings consist of both performance and non-performance-related refactorings. Using the Issues-Related Projects (**PS2**) dataset, we detected and retrieved all of the refactorings identified by RefactoringMiner within each project's repository. This was achieved by passing each project's GitHub URL and default branch to RefactoringMiner, which then clones the project's repository and extracts all refactorings from the default branch. We used the projects default branches to extract refactorings, as they are easily accessible through the GitHub API. The default branch is typically the most comprehensive source of a project's commit history, as it includes all changes that have been merged from other branches. Additionally, it mostly serves as the primary point for integrating changes or pull requests related to issues that matter to the project maintainers. Therefore, by focusing on the default branches, we can capture important refactorings across a project's development lifecycle. This approach ensures that we obtain a complete and accurate record of each project's refactorings. We extracted detailed information from both project and refactoring such as *(1) repository fullname, (2) commit id, (3) refactoring types, (4) description and (5) changes in diffs which includes; file path, start line, end line, code elements and code elements types.* In total, we detected 22,763,791 project refactorings.

2. **RC2 Detecting Performance-Related Refactorings:** To focus on performance-related refactorings, we employed a multi-step process leveraging the data obtained from our previous analysis (**GC2**). We used the pull request numbers associated with previously identified performance-related issues as inputs for RefactoringMiner. For

Table 3.3 Overview of Performance-Related Issues and Pull Requests

| **Metric** | **Description** | **Count** |
|---|---|---|
| Initial Issues Identified | Total number of performance-related issues initially identified | 67,184 |
| Issues After Deduplication | Number of unique performance-related issues after removing duplicates | 25,462 |
| Issues With Pull Request | Number of unique performance-related issues with pull request | 22,527 |
| Issues Without Pull Request | Number of unique performance-related issues without pull request | 2,935 |
| Issues Linked With Performance Refactorings | Number of performance-related issues that are linked to at least one performance refactoring | 940 |

each pull request, we extracted a comprehensive set of refactoring features, mirroring the approach used in **RC1**. This initial extraction yielded a substantial dataset of 613,996 potentially performance-related refactorings. To refine this dataset and ensure its relevance, we conducted a thorough comparative analysis between these performance-related refactorings and the broader set of project refactorings. This comparative analysis allowed us to identify 45,504 confirmed performance-related refactorings across 491 distinct projects. This step was crucial in filtering out refactorings that, while potentially performance-related, did not align with the actual refactoring activities within the projects. To further enhance the precision of our dataset, we performed an additional layer of analysis. We compared the commit IDs of the performance-related refactorings found in the project refactorings against those in our performance-related issues dataset. This step was designed to isolate refactorings that were not only performance-related but also directly addressed reported issues and were successfully merged into the projects' main repositories. This rigorous filtering process resulted in the identification of 19,646 high-confidence performance-related refactorings. These refactorings spanned 255 projects and addressed 940 distinct performance-related issues. This refined dataset represents a valuable resource for understanding the practical application of performance-oriented refactoring in real-world software development contexts. Given the importance of these data, we classified the projects containing these verified performance-related refactorings into a separate dataset. We labeled this collection as Refactorings-Related Projects (**PS3**), as detailed in Section 3.2. This categorization facilitates further analysis of projects that have demonstrably engaged in performance-driven refactoring practices.

To better understand why some extracted performance-related refactorings are absent from the project refactorings we randomly selected 50 performance-related issues and manually analyzed the commits in their pull requests. We observed several potential reasons why some performance-related refactorings cannot be found in the project refactorings dataset or why some performance refactorings in the project refactorings dataset do not have corresponding commits in the performance-related issues dataset:

- Forked Repositories: Repository forks might contribute to performance improvements. These contributions might not be merged into the main repository straightforwardly, leading to discrepancies in the refactoring history. Several merge strategies may be deployed in this scenario which include; (1) deleting the branch post-merge, where the commit remains in the repository's history, but is no longer associated with an active branch; (2) the pull request was squashed and merged. Consequently, the original commits from the pull request are no longer part of any branch in the repository; and (3) the pull request was rebased and merged, altering the original commit hashes and causing the original commits to appear as if they do not belong to any branch.
- Different Branches: Performance refactorings might be performed in feature branches, hotfix branches, or other branches that aren't immediately merged into the default branch. Tracking these changes requires monitoring all relevant branches, not just the default one.
- Merge Conflicts and Overwrites: Merges can overwrite or integrate changes in ways that obscure the original intent of individual commits, making it hard to trace performance improvements back to specific issues.
- Commit Granularity: Refactorings can span multiple commits or be bundled with other changes. This makes it difficult to identify the exact commit that addresses a specific performance issue.
- Cross-Referenced Issues: Sometimes, performance refactorings might address multiple issues, or multiple refactorings might address a single issue. Tracking such cross-references can be complex without detailed and consistent documentation practices.
- Nested Repositories: Large projects might include submodules or dependencies that contain significant portions of the codebase. Performance refactorings in these submodules might not be captured if the extraction process only considers the main repository.

- Large Commits: Monolithic commits that include numerous changes can obscure the intent of individual refactorings. Parsing such commits to identify specific performance-related changes can be challenging.

We filter out these issues because ensuring their usefulness is beyond the scope of this work.

3. **RC3. Detecting Non-Performance-Related Refactorings:** To establish a control group for comparison with our performance-related refactorings, we systematically identified non-performance-related refactorings from the same projects. Using our refactoring-related projects (**PS3**), we implemented a two-step filtering process. First, we leveraged the comprehensive set of project refactorings collected during **RC1**. Second, we performed a commit-level comparison between performance-related refactorings and the complete refactoring dataset. Specifically, we extracted all refactorings whose commit IDs did not match those of performance-related refactorings, classifying them as non-performance-related. This methodical separation ensured mutually exclusive datasets while maintaining comparable contexts (e.g., development strategies, project types, developer expertise) across both refactoring categories. Table 3.3 presents an overview of the data collected for performance-related issues and pull requests, while Table 3.4 summarizes the distribution of both types of refactorings identified within our studied repositories.

## 3.3 Study Results

In this section, we present the motivation, approach, and the results of each of our five research questions:

**RQ1. What is the likelihood of finding refactoring occurrences in performance-related commits compared to non-performance-related commits?**

**Motivation:**

Addressing performance-related issues in software systems presents a range of challenges and complexities for developers [1]. This research question aims to understand whether developers approach performance-related issues differently compared to non-performance-related

Table 3.4 Overview of Performance and Non-Performance-Related Refactorings

| Metric | Description | Count |
|---|---|---|
| Project Refactorings | Total number of refactorings identified in the repositories of the issues-related projects **PS2** | 22,763,791 |
| Initial Performance Refactorings | Total number of performance-related refactorings identified using performance-related issues | 613,996 |
| Performance Refactorings In Projects | Total number of performance-related refactorings initially identified in project refactorings | 45,504 |
| Non-Performance Refactorings In Projects | Total number of non-performance-related refactorings initially identified in project refactorings | 22,718,287 |
| Performance Refactorings | Number of performance-related refactorings identified in project refactorings and has their corresponding commit in performance-related issues | 19,646 |
| Non-Performance Refactorings | Number of non-performance refactorings that are associated to the refactoring-related projects **PS3**. | 8,273,491 |

issues. Specifically, we seek to determine the likelihood of refactoring in performance-related commits versus non-performance-related commits. Understanding this difference is crucial for several reasons. Insights into the development practices adopted by software engineers when addressing performance issues can inform better project management and planning. If refactoring is more prevalent in performance-related commits, it may indicate that developers frequently need to restructure code to enhance performance. This suggests that performance improvements often require substantial codebase changes beyond simple optimizations. Recognizing that performance improvements might involve significant refactoring can lead to more accurate time estimations and prioritization of tasks. This knowledge can help project maintainers allocate resources more effectively and set realistic deadlines, ultimately improving the efficiency and success of software development projects.

**Approach:**

Our investigation began with a comprehensive analysis of commits related to both performance-related refactorings and their associated performance-related issues. The process involved several key steps:

1. **Data Extraction and Categorization:** To find refactorings that occurred in performance-

related commits, we analyzed two primary sources: (1) Performance-Related Refactorings: We extracted details, including *repository fullnames, commit ids, issue numbers, and pull request numbers*. This allowed us to identify unique commit IDs associated with performance-related refactorings identified in **RC2**. (2) Performance-Related Issues: We used the commit IDs from the performance-related refactorings to retrieve performance-related issues (**GC1**) related to the identified performance-related refactorings.

2. **Comparative Analysis to Categorize Performance-Related Refactorings:** We conducted a detailed comparative analysis between the commits identified as performance-related refactorings and those from the associated pull requests. This step was critical in distinguishing which specific commits contained refactorings and which did not. Firstly, we calculated the total number of unique commit IDs associated with performance-related refactorings. We followed up to determine the number of performance issues related to these refactorings, and then we extracted and counted the commit IDs linked to each performance-related issue. Furthermore, we compared the set of commit IDs from performance-related issues with those from performance-related refactorings. Finally, we quantified the overlap between these two sets to understand how many performance issue commits were directly associated with refactoring efforts. Our analysis revealed: 2,316 commit IDs containing performance-related refactorings and 3,388 commit IDs without performance-related refactorings. This provides insights into the prevalence and distribution of refactoring activities within performance-related issue resolution processes.

3. **Comparative Analysis to Categorize Non-Performance-Related Refactorings:** We extend our analysis to include non-performance-related refactorings collected in **RC3**. This step was essential for establishing a baseline for comparison and gaining a deeper understanding of the broader refactoring landscape. We specifically analyzed the *commit_id* key for each commit in this dataset to determine whether a refactoring was associated with each commit. Our study of 2,199,493 commits revealed that 612,796 commit IDs had at least one refactoring, while 1,586,697 did not. Detailed results are available in our replication package.

4. **Statistical Analysis and Hypothesis Testing using Chi-Test of Independence:** We performed the Chi-square Test of Independence [80] to determine if there is a statistically significant association between the type of commit (performance-related or non-performance-related) and the presence of refactoring as shown in Equations 3.1 and 3.2. We set a null hypothesis($H_0$) that *there is no association between commit*

*type and refactoring presence* and also an alternative hypothesis($H_1$) that *there is an association between commit type and refactoring presence.* We establish the significance level of our test at 0.05. If the p-value obtained from our test is less than this significance level, we would reject the null hypothesis and accept the alternative hypothesis, concluding that there is a statistically significant association between commit type and refactoring presence. This approach ensures that our findings are robust and reliable, providing meaningful insights into the relationship between commit types and refactoring activities. We also calculate the odds ratio [81] to quantify the strength of the association between performance-related commits and the presence of refactoring as shown in Equation 3.3. An odds ratio greater than 1 indicates a higher likelihood of refactoring in performance-related commits compared to non-performance-related commits.

$$\chi^2 = \sum \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \tag{3.1}$$

*$O_{ij}$ denotes the observed frequency of each cell in Table 3.5, where $i$ indicates the row (either performance-related or non-performance-related) and $j$ indicates the column (either refactoring present or refactoring absent).*

$$E_{ij} = \frac{(\text{Row Total}_i) \times (\text{Column Total}_j)}{\text{Grand Total}} \tag{3.2}$$

*$E_{ij}$ is the expected frequency for each cell, calculated under the assumption of independence between commit types and the presence of refactoring.*

$$\text{Odds Ratio (OR)} = \frac{a \cdot d}{b \cdot c} \tag{3.3}$$

where $a$, $b$, $c$, and $d$ represent different commit counts related to the presence or absence of refactoring in performance-related and non-performance-related commits. (See Appendix A for detailed explanations of these variables.)

**Results:**

Table 3.5 shows an overview of the prevalence of refactoring in performance-related and non-performance-related commits. Based on our statistical analysis, we obtained a Chi-Square statistic of 458.64, with a p-value of approximately 9.51e-102. The degree of freedom is 1, and we calculated an odds ratio of 1.77. The extremely low p-value indicates

Table 3.5 Prevalence of Refactorings in Performance-Related and Non-Performance-Related Commits

| Commit Type | Refac. Present | Refac. Absent | Pres.% | Abs.% | Total |
|---|---|---|---|---|---|
| Perf-Related | 2,316 | 3,388 | 40.6 | 59.4 | 5,704 |
| Non-Perf-Related | 612,796 | 1,586,697 | 27.9 | 72.1 | 2,199,493 |

Note: Refac. = Refactoring, Pres. = Present, Abs. = Absent, Perf-Related = Performance-Related Commits, Non-Perf-Related = Non Performance-Related Commits

a statistically significant association between the type of commit (performance-related or non-performance-related) and the presence of refactoring. Consequently, we reject our null hypothesis ($H_0$) in favor of our alternative hypothesis ($H_1$). Interestingly, the association, as interpreted through the odds ratio, suggests that refactoring is more likely in performance-related commits compared to non-performance-related ones. The odds ratio of 1.77 indicates that performance-related commits are about 1.77 times more likely to include refactoring compared to non-performance-related commits.

*Our analysis reveals that while refactoring occurs in both types of commits, it is significantly more frequent in the context of resolving performance issues (odds ratio of 1.77). This indicates that developers are more likely to undertake refactoring when addressing performance issues compared to other types of issues. This trend suggests that refactoring may serve as a strategy to improve performance. Developers might focus on simplifying complex algorithms, eliminating redundancies, or optimizing the overall architecture, all of which could contribute to enhanced efficiency and speed. These actions are often driven by the need to address performance bottlenecks, though other factors, such as code readability or maintainability, could also play a role in the decision to refactor.*

**RQ2. What refactoring types are most commonly used when addressing performance issues?**

**Motivation:**

In software systems, resolving performance-related issues is a critical aspect of ensuring optimal system functionality and user experience [2]. Performance issues generally impact overall software performance, resulting in slow response times and increased resource consumption, leading to a degradation of user experience [25, 82]. To tackle these challenges, developers often resort to various strategies, some of which can involve using code refactoring to improve

performance [20]. Knowledge of refactoring strategies proven to work on past performance issues could empower developers to improve the speed at which they resolve future performance issues. Indeed, studies of general refactoring prevalence have been conducted in the past [58, 83]. However, to the best of our knowledge, prior work has yet to identify which refactoring types (if any) are generally used to resolve performance issues. Therefore, in this research question, we aim to understand the specific refactoring types used by developers to resolve performance-related issues.

**Approach:**

To find the refactoring types most commonly used in addressing performance-related issues across our projects, we implemented a comprehensive analytical approach. This method combines descriptive statistics with inferential statistical testing to provide robust insights into refactoring practices.

1. **Data Preparation and Descriptive Analysis:** We began by refining our performance-related refactorings dataset, focusing on two key attributes: *projects* and *refactoring types*. This refinement enabled us to calculate the frequency of each refactoring type across our study projects, forming the basis for our analysis. To normalize our findings across projects with varying sizes and refactoring intensities, we computed the average frequency of each refactoring type using Equation 3.4, ensuring that projects with zero occurrences were included to provide a more accurate representation of the overall distribution.

$$\text{average\_frequency}(t) = \frac{1}{N}\sum_{i=1}^{N}\text{freq}_{i,t} \tag{3.4}$$

*Where $N$ represents the total number of projects in the study, and $freq_{i,t}$ represents the frequency of occurrence of refactoring type $t$ in project $i$, with $freq_{i,t} = 0$ for projects without that refactoring type.*

2. **Statistical Analysis of Refactoring Type Frequencies:** To analyze whether certain refactoring types are used more frequently than others when addressing performance issues, we applied the Friedman Test. This non-parametric test is specifically suited for dependent samples, where multiple related observations (i.e., different refactoring types) are measured across the same projects. In our context, the same set of projects contains various types of refactorings, making the observations inherently dependent. This dependency invalidates the assumptions of tests like Kruskal-Wallis,

which assume group independence. We structured our data as a matrix where each row represents a project, and each column corresponds to a unique refactoring type. The matrix entries denote the frequency with which each refactoring type appears within each project. The Friedman test evaluates the null hypothesis ($H_0$) that *the distribution of refactoring type frequencies is the same across all types.* The alternative hypothesis ($H_1$) suggests that *at least one refactoring type is used with significantly different frequency*

$$\chi_F^2 = \frac{12}{nk(k+1)} \sum_{j=1}^{k} R_j^2 - 3n(k+1) \tag{3.5}$$

Where $n$ is the number of projects, $k$ is the number of refactoring types and $R_j$ is the sum of ranks for refactoring types $j$ across projects.

3. **Scott-Knott Effect Size Difference Test:**

   Following the Friedman test, we employed the Scott-Knott Effect Size Difference Test to identify which specific refactoring types differ significantly in usage. This allowed us to group refactoring types into statistically distinguishable clusters of usage frequency. This decision was driven by the need for a more refined approach than traditional post-hoc tests, which often suffer from issues such as inflated Type I error rates and limited interpretability when performing multiple comparisons. We apply Equation 3.6 to compute the test statistic and corresponding p-value for the Scott-Knott test. This methodology allows us to move beyond raw frequency counts, enabling a more structured and interpretable analysis of which refactoring types are commonly employed when addressing performance issues. Full details of the test formulation are provided in Appendix A.

$$\lambda = \frac{\pi}{2} \cdot (\pi - 2) \cdot \frac{B_o}{\sigma_o^2} \tag{3.6}$$

Where $\lambda$ is asymptotically a $\chi^2$ distributed random variable with $\nu_o = k(\pi - 2)$ degrees of freedom. (See Appendix A for detailed explanations of variables.)

We used the ScottKnott package in R [84] for our analysis. We employed the `SK` function for clustering the treatment means of our main factor (refactoring types). The significance level ($\alpha$) used was set to 0.05. The results of the Scott-Knott Test are presented as distinct groups (labelled as *High-Frequency Refactorings* and *Low-Frequency Refactorings*).

- **High-Frequency Refactorings:** This group comprises refactoring types with higher mean ranks.
- **Low-Frequency Refactorings:** This group includes refactoring types with lower mean ranks.

By combining the Friedman Test with the Scott-Knott Test, we provide a comprehensive statistical analysis that identifies significant differences in refactoring types across projects and offers a meaningful and interpretable grouping of refactoring types. This approach enables us to draw more nuanced conclusions about the patterns and preferences in addressing performance issues across different software development contexts.

**Results:**

Table 3.6 shows an overall overview of the frequencies and average frequencies of the refactoring types analysed from our performance-related refactorings. The analysis provided insights into the frequency of various refactoring types used to address performance issues across our study projects.

1. **Frequency Results:** “Change Variable Type", alongside “Add Parameter" and “Move Class", reveal key insights into software refactoring practices. These top refactoring types primarily relate to code structure, type manipulation, and method modifications.

   The high frequency of “Move Class" refactorings suggests that changes to class structure may be related to addressing performance issues. For example, in the *product-apim* project[2], developers applied the "Move Class" refactoring to the `AuthenticatorClient` class. This refactoring was part of addressing a performance issue where the API Management (APIM) system became slow and unresponsive, with all synapse threads waiting on the `PassThroughMessageProcessor`. By moving the class to its correct package, developers aimed to avoid class loading conflicts, reduce the size of individual packages, and enhance the overall organization of the code. While these changes may have improved code clarity and potentially addressed performance bottlenecks, it is important to note that other factors, such as code organization and maintainability, could also contribute to the decision to apply such refactorings.

   “Change Variable Type" exemplifies low-level code adjustments that can significantly impact performance. In the *egeria-connector-ibm-information-server* project (commit:

[2]https://github.com/wso2/product-apim/commit/2f35cddd1f8159a375c9c307e7648aaa87f81da7

Table 3.6 Distribution of refactoring types across projects. Full results available in the replication package.

| **Most Frequent Refactoring Types** | | |
|---|---|---|
| Refactoring Types | Frequency | Average Frequency |
| Change Variable Type | 1807 | 7.09 |
| Add Parameter | 1215 | 4.76 |
| Move Class | 929 | 3.64 |
| Change Parameter Type | 880 | 3.45 |
| Rename Method | 875 | 3.43 |
| Add Method Annotation | 796 | 3.12 |
| Change Return Type | 760 | 2.98 |
| Rename Variable | 746 | 2.93 |
| Change Attribute Type | 676 | 2.65 |
| Extract Method | 653 | 2.56 |
| **Least Frequent Refactoring Types** | | |
| Refactoring Types | Frequency | Average Frequency |
| Inline Attribute | 5 | 0.02 |
| Merge Method | 3 | 0.01 |
| Replace Attribute | 2 | 0.01 |
| Replace Anonymous With Class | 2 | 0.01 |
| Merge Class | 2 | 0.01 |
| Split Class | 2 | 0.01 |
| Merge Catch | 2 | 0.01 |
| Assert Throws | 2 | 0.01 |
| Parameterize Test | 1 | 0.00 |
| Modify Variable Annotation | 1 | 0.00 |

31979550)[3], developers modified the `Identity` class by changing a variable type from `String[]` to `List<String>`. This refactoring addressed search bugs and improved the efficiency of handling identity components and tokens by leveraging more flexible and optimized data structures.

Other frequent refactoring types include “Change Parameter Type", “Rename Method", and “Add Method Annotation", indicating the dynamic nature of code maintenance and optimization.

In contrast, the least frequent refactoring types such as “Inline Attribute", “Merge Method", and several others with just 1-2 occurrences (“Replace Attribute", “Replace Anonymous With Class", “Merge Class", “Split Class", “Merge Catch", “Assert Throws", “Parameterize Test", and “Modify Variable Annotation") suggest these are highly specialized refactoring techniques. Their infrequency implies that while valid, they are used sparingly and may address very specific, nuanced performance or structural issues in particular project contexts.

2. **Friedman Test Results:** The results of the Friedman Test reveal a *test statistic of 5129.10* and a *p-value of <0.001*. This significant result indicates that there are statistically significant differences in the frequencies of at least some refactoring types across the analyzed projects. The test allows us to confidently reject the null hypothesis that *the distributions of refactoring type frequencies are identical across all types*.

   These findings underscore the variability in how different refactoring strategies are applied when addressing performance issues. It reinforces the idea that not all refactoring types are used uniformly and that their application may be influenced by project-specific factors. While this study does not attempt to explain all underlying causes, it highlights the importance of recognizing such variation. To further interpret these differences, we apply a post-hoc clustering analysis using the Scott-Knott test to identify distinct groups of commonly used refactoring strategies.

3. **Scott-Knott Post-Hoc Analysis:** While the Friedman test confirmed the presence of statistically significant differences in refactoring type frequencies, it does not reveal which specific refactoring types differ or how they relate in terms of usage.

   The Scott-Knott algorithm partitioned the refactoring types into two statistically distinct and non-overlapping groups, which we designate as *High-Frequency Refactorings* and *Low-Frequency Refactorings*, as summarized in Table 3.7. This clustering provides a clearer view of usage patterns, enabling us to distinguish between commonly used

[3]https://github.com/odpi/egeria-connector-ibm-information-server/commit/31979550146ee95700969edf547d610dfc28ae4

Table 3.7 Refactoring Types with Frequency Group

| Refactoring Type | Grp | Refactoring Type | Grp |
|---|---|---|---|
| Modify Parameter Annotation | HFR | Remove Parameter Annotation | LFR |
| Pull Up Method | HFR | Remove Variable Modifier | LFR |
| Change Variable Type | HFR | Move Code | LFR |
| Pull Up Attribute | HFR | Parameterize Variable | LFR |
| Add Method Annotation | HFR | Add Method Modifier | LFR |
| Move Method | HFR | Replace Pipeline With Loop | LFR |
| Merge Method | HFR | Move And Rename Class | LFR |
| Change Parameter Type | HFR | Inline Method | LFR |
| Add Parameter | HFR | Replace Attribute With Variable | LFR |
| Add Parameter Modifier | HFR | Split Variable | LFR |
| Change Attribute Type | HFR | Extract Attribute | LFR |
| Change Return Type | HFR | Replace Variable With Attribute | LFR |
| Add Thrown Exception Type | HFR | Remove Variable Annotation | LFR |
| Move Class | HFR | Encapsulate Attribute | LFR |
| Rename Method | HFR | Extract Class | LFR |
| Move Attribute | HFR | Split Conditional | LFR |
| Rename Parameter | HFR | Replace Anonymous With Lambda | LFR |
| Add Attribute Annotation | HFR | Change Class Access Modifier | LFR |
| Change Method Access Modifier | HFR | Merge Attribute | LFR |
| Remove Parameter | HFR | Add Class Modifier | LFR |
| Rename Variable | HFR | Extract Superclass | LFR |
| Rename Attribute | HFR | Modify Method Annotation | LFR |
| Push Down Method | HFR | Merge Conditional | LFR |
| Add Parameter Annotation | HFR | Localize Parameter | LFR |
| Change Thrown Exception Type | HFR | Move And Rename Attribute | LFR |
| Extract Method | HFR | Merge Parameter | LFR |
| Add Variable Modifier | HFR | Move And Rename Method | LFR |
| Add Variable Annotation | HFR | Reorder Parameter | LFR |
| Change Attribute Access Modifier | HFR | Collapse Hierarchy | LFR |
| Modify Class Annotation | HFR | Merge Variable | LFR |
| Add Attribute Modifier | HFR | Invert Condition | LFR |
| Remove Parameter Modifier | HFR | Change Type Declaration Kind | LFR |
| Move And Inline Method | HFR | Split Method | LFR |
| Extract And Move Method | HFR | Extract Interface | LFR |
| Remove Attribute Modifier | HFR | Parameterize Attribute | LFR |
| Remove Attribute Annotation | HFR | Inline Attribute | LFR |
| Remove Class Annotation | HFR | Remove Class Modifier | LFR |
| Add Class Annotation | HFR | Move Source Folder | LFR |
| Replace Attribute | HFR | Split Attribute | LFR |
| Extract Variable | HFR | Split Parameter | LFR |
| Remove Method Annotation | HFR | Extract Subclass | LFR |
| Modify Attribute Annotation | HFR | Replace Loop With Pipeline | LFR |
| Rename Class | HFR | Merge Catch | LFR |
| Push Down Attribute | HFR | Split Class | LFR |
| Remove Thrown Exception Type | HFR | Assert Throws | LFR |
| Inline Variable | HFR | Replace Anonymous With Class | LFR |
| Remove Method Modifier | HFR | Parameterize Test | LFR |
| | | Merge Class | LFR |
| | | Modify Variable Annotation | LFR |

Grp: Group, HFR: High-Frequency Refactorings, LFR: Low-Frequency Refactorings

refactoring strategies and those that are applied less frequently across the analyzed projects.

- The separation into two groups suggests that refactoring types in the *High-Frequency Refactorings group* may be more commonly associated with addressing performance issues.
- Many refactoring types in the *High-Frequency Refactorings group* (e.g., “Pull Up Method," “Move Method," “Pull Up Attribute") involve structural changes to code, often related to class hierarchies and method organization. This indicates that developers frequently apply these refactorings in the context of performance-related issues, though the specific impact on performance may vary depending on the context of each change.
- The high ranking of “Modify Parameter Annotation", “Add Method Annotation",

and “Change Variable Type" indicates a focus on fine-tuning method signatures and variable declarations. This may *reflect efforts to optimize method invocations and data handling, which can have performance implications.*

- Refactoring types in the Low-Frequency Refactorings group, such as “Extract Class," “Extract Superclass," and “Extract Interface," appear to be less frequently applied in the context of addressing performance issues. This could be due to their potentially higher complexity or risk of introducing performance regressions.
- The results suggest a preference for method-level refactorings (e.g., “Merge Method") as observed in the *High-Frequency Refactorings* group over class-level refactorings (e.g., “Merge Class") observed in the *Low-Frequency Refactorings* group when addressing performance issues. *This may indicate that developers find it more effective or less risky to optimize at the method level rather than making larger architectural changes.*

*Our results indicate that performance-related issues in software development are often tackled with altering class structures and enhancing method configurations. “Move Class" as the most frequent refactoring indicates a common need to optimize the location of classes within the application’s architecture for better performance, possibly reducing coupling and improving cohesion. The significantly lower frequency of refactorings like “Split Class" and “Merge Class" suggests that these actions, while potentially beneficial for readability and maintainability, might not be as critical for performance issue resolution, or their need arises in more specific contexts.*

**RQ3. Do performance issues affect the choice of refactoring types used by developers?**

**Motivation:**

Performance is a critical aspect of software systems, and optimizing performance-related issues is essential for achieving efficient and responsive software behaviour. However, refactoring is not limited to performance improvements; it also plays a crucial role in maintaining and enhancing the overall quality of code. By examining the distribution of refactorings related to performance issues and comparing it to the distribution of all other refactorings for each of our selected projects, we seek to gain insights into whether developers use different refactoring types when facing performance-related issues. By differentiating between performance

and non-performance-related refactorings, we aim to provide an understanding of the specific refactoring types that are most prevalent in performance-related and non-performance-related refactorings of our subject systems. By identifying the differences between these two refactorings we aim to provide a better understanding of the preferred strategies for dealing with performance-related issues and how they differ from all other refactorings aimed at enhancing other aspects of software systems.

**Approach:**

1. **Statistical Analysis and Hypothesis Testing using Chi-Square Goodness-of-Fit Test:** To address our research question, we conduct a comprehensive analysis of the performance-related refactorings and non-performance-related refactorings datasets **RC2 and RC3**. Our analytical approach involves aggregating the frequency sums for each refactoring type across both datasets, encompassing all projects under investigation **PS3**. This aggregation provides a holistic view of refactoring patterns in performance-related and non-performance-related contexts. To rigorously assess the statistical significance of any observed differences between these two refactoring categories, we employ the Chi-Square Goodness-of-Fit Test. This statistical method is applied to each refactoring type present in our dataset, as formulated in Equation 3.7. The test is designed to evaluate the following null hypothesis ($H_0$):

   > $H_0$: *The distribution of a refactoring type in performance-related refactorings is equivalent to that in non-performance-related refactorings.*

   We also calculate the odds ratio to quantify the strength of the association for each refactoring type between performance-related and non-performance-related refactorings as shown in Equation 3.6.

   We use the chi-square test to analyze the distribution of refactoring types in performance-related refactorings:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i} \tag{3.7}$$

   Where $\chi^2$ is the chi-square statistic, $O_i$ is the observed frequency, and $E_i$ is the expected frequency of the $i$-th refactoring type. (See Appendix A for detailed explanations.)

Table 3.8 Frequency and Statistical Analysis of Different Refactoring Types in Performance-Related and Non-Performance-Related Refactorings

| | **Refactoring Types** | **Perf. Freq** | **Non-Perf. Freq** | **Perf. %** | **Non-Perf. %** | $\chi^2$ | **p-value** | **OR** |
|---|---|---|---|---|---|---|---|---|
| Most Frequent | Change Variable Type | 1807 | 654350 | 9.20 | 7.91 | 44.50 | <0.001 | 1.18 |
| | Add Parameter | 1215 | 382370 | 6.18 | 4.62 | 108.16 | <0.001 | 1.36 |
| | Move Class | 929 | 320860 | 4.73 | 3.88 | 37.79 | <0.001 | 1.23 |
| | Change Parameter Type | 880 | 496108 | 4.48 | 6.00 | 79.80 | <0.001 | 0.74 |
| | Rename Method | 875 | 366831 | 4.45 | 4.43 | 0.01 | 0.905 | 1.01 |
| | Add Method Annotation | 796 | 496478 | 4.05 | 6.00 | 131.75 | <0.001 | 0.66 |
| | Change Return Type | 760 | 400498 | 3.87 | 4.84 | 40.03 | <0.001 | 0.79 |
| | Rename Variable | 746 | 280483 | 3.80 | 3.39 | 9.79 | 0.002 | 1.13 |
| | Change Attribute Type | 676 | 270550 | 3.44 | 3.27 | 1.75 | 0.185 | 1.05 |
| | Extract Method | 653 | 209031 | 3.32 | 2.53 | 50.23 | <0.001 | 1.33 |
| Least Frequent | Merge Class | 2 | 1680 | 0.01 | 0.02 | 0.55 | 0.456 | 0.50 |
| | Replace Attribute | 2 | 571 | 0.01 | 0.01 | 0.02 | 0.902 | 1.48 |
| | Modify Variable Annotation | 1 | 95 | 0.01 | 0.00 | 0.33 | 0.567 | 4.43 |
| | Parameterize Test | 1 | 99 | 0.01 | 0.00 | 0.29 | 0.588 | 4.25 |
| | Replace Generic With Diamond | 0 | 83996 | 0.00 | 1.02 | 200.48 | <0.001 | 0.00 |
| | Try With Resources | 0 | 8375 | 0.00 | 0.10 | 18.92 | <0.001 | 0.00 |
| | Move Package | 0 | 5634 | 0.00 | 0.07 | 12.40 | <0.001 | 0.00 |
| | Rename Package | 0 | 5013 | 0.00 | 0.06 | 10.93 | <0.001 | 0.00 |
| | Split Package | 0 | 1493 | 0.00 | 0.02 | 2.61 | 0.106 | 0.00 |
| | Merge Package | 0 | 1182 | 0.00 | 0.01 | 1.89 | 0.169 | 0.00 |

***Keys:Perf.Freq:****Frequency in Performance-Related Refactorings* ,***Non-Perf.Freq:****Frequency in Non-Performance-Related Refactorings* ,***Perf.%:****Percentage of Refactoring Types in Performance-Related Refactorings* ,***Non-Perf.%:****Percentage of Refactoring Types in Non-Performance-Related Refactorings* $\chi^2$*:Chi-Square Statistic* ,***p-value:****P-value* ,***OR:****Odds Ratio*

2. **Calculate Percentage of Refactoring Types:** Furthermore, we calculate the percentage of each refactoring type by dividing its frequency by the total frequency of refactorings in each category. The formula for the percentage is shown in Equation 3.8.

$$\text{percentage} = \frac{\text{Frequency of Refactoring Type}}{\text{Total Frequency of Refactorings}} \times 100 \tag{3.8}$$

3. **Calculate Cohen's D:** Finally, to further enrich our analysis and provide a more comprehensive understanding of the differences between performance-related and non-performance-related refactorings across all the projects, we employ Cohen's D test. This statistical measure, as formulated in Equations 3.9 through to 3.11, allows us to quantify the effect size of the distributions between the performance-related refactorings and non-performance-related refactorings datasets. A detailed description of this test can be found in Appendix A

$$\textit{Cohen's } d = \frac{M_1 - M_2}{SD} \tag{3.9}$$

Where $M_1$ and $M_2$ are the means of non-performance-related and performance-related refactoring frequencies across all projects, respectively, and $SD$ is the pooled standard deviation. (See Appendix A for detailed explanations.)

Mean Frequency ($M_1$ and $M_2$):

$$\begin{aligned} M_1 &= \frac{\sum \text{Non-Performance Freq}}{N_1} \\ M_2 &= \frac{\sum \text{Performance Freq}}{N_2} \end{aligned} \tag{3.10}$$

*Where $\sum$ Non-Performance Freq is the sum of frequencies for non-performance-related refactorings, $\sum$ Performance Freq is the sum of frequencies for performance-related refactorings, $N_1$ is the number of non-performance-related refactorings, and $N_2$ is the number of performance-related refactorings.*

Pooled Standard Deviation (SD):

$$SD = \sqrt{\frac{(N_1 - 1) \cdot SD_1^2 + (N_2 - 1) \cdot SD_2^2}{N_1 + N_2 - 2}} \tag{3.11}$$

*Where $SD_1$ is the standard deviation of non-performance-related refactorings and $SD_2$ is the standard deviation of performance-related refactorings.*

### Results

We present a detailed analysis of the Chi-Square Goodness-of-Fit Test results and the corresponding Odd Ratio values for the top 1044 most frequent and least frequent refactoring types observed in our performance-related refactoring dataset identified in **RC2**. Table 3.8

provides a summary of our results. A full analysis of all of our collected refactoring types can be found in our replication package. This analysis provides insights into the distribution and association of refactoring types between performance-related and non-performance-related refactorings. Our comprehensive analysis reveals several key insights:

1. **Statistical Significance and Effect Sizes** Our study revealed diverse patterns across refactoring types, with many showing statistically significant differences ($p < 0.05$) between performance-related and non-performance-related contexts, and varying effect sizes as measured by odds ratios.

    - **Statistical Significance:** The low p-values ($< 0.001$ for most types) indicate that the observed differences in refactoring distributions are unlikely to be due to chance. This suggests that there are indeed systematic differences in how refactorings are applied in performance-related versus non-performance-related contexts.
    - **Effect Sizes:** Some refactorings showed a higher likelihood in performance-related contexts. For instance, “Add Parameter" (OR = 1.36), “Extract Method" (OR = 1.33) and “Move Class" (OR = 1.23) were more likely to occur in performance-related refactorings. Others were less likely in performance-related contexts, such as “Add Method Annotation" (OR = 0.66) and “Change Return Type" (OR = 0.79).

2. **Most Frequent Refactoring Types**

    - “Add Parameter" ($\chi^2 = 108.16$, $p < 0.001$, OR = 1.36) With a higher Chi-square value and odds ratio, this refactoring shows a more substantial difference. Developers are 36% more likely to add parameters in performance-related contexts, perhaps to pass additional information needed for optimization.
    - “Extract Method" ($\chi^2 = 50.23$, $p < 0.001$, OR = 1.33) The significant difference and odds ratio suggest that method extraction is 32.6% more likely in performance-related contexts. Developers could be more strategic about breaking down methods when performance is a concern.
    - “Move Class" ($\chi^2 = 37.79$, $p < 0.001$, OR = 1.23) The significant difference and odds ratio suggest that class organization is influenced by performance considerations. Developers are 23% more likely to relocate classes when dealing with performance-critical code.

- "Change Variable Type" ($\chi^2 = 44.50$, $p < 0.001$, OR = 1.18) This refactoring shows a significant difference between performance-related and non-performance-related contexts. The odds ratio suggests that developers are 17.9% more likely to change variable types in performance-related code, possibly to optimize memory usage or computation speed.
- "Rename Variable" ($\chi^2 = 9.79$, $p = 0.002$, OR = 1.13) While statistically significant, this refactoring shows a weaker association with context compared to others. Variable renaming is 12.5% more likely in performance-related contexts, but readability likely remains the primary concern.

3. **Least Frequent Refactoring Types**

   - The following four refactoring types were never observed in performance-related contexts in our datasets: "Replace Generic With Diamond", "Try With Resources", "Move Package", "Rename Package". This might be because developers fix issues such as package naming, package structure, and resource handling may not typically be prioritized when addressing performance issues. It could be that these types of refactorings are more relevant in other contexts, such as code readability or maintainability, rather than directly addressing performance issues.

*Our results indicate that performance issues indeed appear to affect the choice of refactoring types used by developers. Some refactoring types, particularly those related to method signatures and annotations (e.g., Add Method Annotation, Change Parameter Type), are much more prevalent in performance-related activities, while structural refactorings (e.g., Move Class, Extract Method) are much less prevalent, suggesting that developers may need different types of tooling support (e.g., refactoring recommendation) when dealing with performance issues. Furthermore, the complete lack of some refactorings types when dealing with performance-related issues (e.g., "Replace Generic With Diamond") is also noteworthy and may warrant further investigation to determine whether developers knowingly handle such refactoring opportunities differently for performance issues.*

Moreover, the results of our analysis on the percentage of each refactoring type reveals three distinct groups:

- **Refactoring Types More Prevalent in Performance-Related Contexts:** This group includes "Change Variable Type" (9.20% vs 7.91%), "Add Parameter" (6.18% vs 4.62%), "Move Class" (4.73% vs 3.88%), and "Extract Method" (3.32% vs 2.53%).

These refactorings occur more frequently in performance-related contexts, indicating that they are often applied when addressing performance issues. The increased frequency suggests that these types of refactorings may be associated with optimizing code in ways that are relevant to improving performance, though the exact impact may vary depending on the context.

- **Refactoring Types More Prevalent in Non-Performance-Related Contexts:** This group comprises “Change Parameter Type" (6.00% vs 4.48%), “Add Method Annotation" (6.00% vs 4.05%), “Replace Generic With Diamond" (1.02% vs 0.00%), and “Try With Resources" (0.10% vs 0.00%). The absence or lower prevalence of some of these refactorings in performance-related scenarios indicates that they are less prioritized when addressing performance issues.

- **Refactoring Types with No Significant Difference Between Both Contexts:** This group includes “Merge Class", “Replace Attribute", “Modify Variable Annotation", and “Parameterize Test". This suggests that these refactorings are applied similarly regardless of performance considerations and might be driven by other factors such as code organization, design preferences, or specific project requirements. Their low overall frequency indicates that they are not commonly used in either context, possibly due to their specialized nature or limited applicability in the codebases studied.

This grouping shows how refactoring types are used in performance-related versus non-performance-related contexts, highlighting the varying priorities and approaches taken by developers when refactoring code.

*The analysis of percentages for the most and least frequent refactoring types reveals important patterns in how refactoring is approached in performance-related scenarios. Refactoring types such as “Change Variable Type" and “Add Parameter" are more prevalent in performance-related refactorings, indicating their importance in addressing performance issues. Conversely, some refactoring types are less relevant or absent in performance-related refactorings such as “Move Package” and “Merge Class”, highlighting the specific nature of addressing performance issues.*

Finally, the computed Cohen’s d value is -0.929, which confirms the insights provided by the difference in magnitude between performance and non-performance-related refactorings: The negative value indicates that, on average, the frequency of performance-related refactorings is lower than that of non-performance-related refactorings. Given $|d| = 0.929$, which exceeds the threshold for a large effect size, we can confirm that there is a substantial difference between

the frequencies of performance-related and non-performance-related refactorings across all the projects. This suggests that developers engage in general code improvement and maintenance activities more often than they undertake refactorings specifically aimed at addressing performance issues. This significant disparity in refactoring frequencies has several potential implications: (a) It may indicate that addressing performance issues is not a primary driver for most refactoring activities. (b) There could be a need for increased awareness or tools to support performance-related refactoring. (c) The lower frequency of performance-related refactorings might reflect the complexity or perceived risk associated with such changes.

*Our findings suggest several avenues for future research in software refactoring and performance optimization. Further studies could explore why performance-related refactorings are less frequent, identifying potential barriers or misconceptions that deter developers. Researchers might also assess the long-term impact of this imbalance on system efficiency and maintainability, providing insights into the trade-offs between different refactoring strategies. Additionally, there is potential to develop strategies or tools that promote more frequent performance-oriented refactoring, leading to better-balanced and more effective software maintenance practices. These directions could enhance our understanding of refactoring's role in maintaining high-performance, maintainable software systems.*

**RQ4. What are the patterns of co-applied refactoring types in performance-related and non-performance-related refactorings?**

**Motivation:**

Refactoring is a sophisticated and multifaceted process that often involves the strategic combination of various refactoring types [12, 26, 27]. Understanding these combinations is crucial for gaining deep insights into developers' decision-making processes, best practices in software maintenance and evolution, and the intricate relationships between different refactoring techniques. The complexity of modern software systems often necessitates a holistic approach to refactoring, where multiple techniques are applied in concert to achieve desired improvements in code quality, maintainability, and performance. To explore the intricacies of these combinations, we employ *Association Rule Mining*, specifically utilizing the *Apriori algorithm* [28]. This data mining technique is particularly well-suited for discovering interesting relationships hidden in large datasets, making it an ideal tool for uncovering patterns in refactoring practices across our studied projects. Our primary objective is to identify, ana-

lyze, and interpret the associations and patterns between co-applied refactoring types, with a particular focus on distinguishing between performance-related and non-performance-related refactorings. This analysis aims to provide a clearer and more nuanced understanding of how developers tend to combine these refactoring types when addressing performance-related issues versus non-performance-related concerns. By examining these patterns, we aim to: (1) Identify frequently co-occurring refactoring types in both performance-related and non-performance-related scenarios. (2) Compare and contrast the refactoring patterns between performance-related and general code improvement efforts. (3) Contribute to the broader understanding of software evolution and maintenance practices in addressing performance issues.

**Approach:**

This RQ employs a systematic approach to uncover patterns of co-applied refactoring types, leveraging advanced data mining techniques and statistical analysis. We focus our investigation on both the performance-related and non-performance-related refactorings datasets as identified in **RC2** and **RC3**.

The primary objective of this analysis is: (a) To identify patterns of co-applied refactoring types in general code refactorings, and (b) To identify specific patterns of refactorings used in addressing performance issues. By examining these two datasets in parallel, we aim to contextualize performance-specific refactoring strategies within the broader landscape of software refactoring practices. We outline our approach in three main steps:

1. **Extract Instances of Performance-related and Non-Performance-Related Refactoring Types:** We begin by extracting data from our performance and non-performance-related refactorings dataset, focusing on two key elements: (1) Commit IDs: These unique identifiers allow us to track specific instances of refactoring in the project's version history. (2) Corresponding refactoring types: For each commit, we record the specific refactoring types applied. This process enables us to map each commit ID to the corresponding refactoring types that were implemented. To organize the data for analysis, we group the refactoring types associated with the same commit ID into sets, referred to as *itemsets* or *transactions*. Each transaction represents a unique instance of refactoring types, where individual refactorings are treated as items. This approach allows us to treat each refactoring commit as a discrete event, consisting of one or more refactoring types, and forms the basis for identifying patterns and associations between different refactorings.

2. **Identify Frequent Itemsets:** To uncover the inherent patterns within our transactions, our next step is to identify frequent itemsets. We employ the Apriori algorithm [28], a seminal model in association rule learning, which is particularly well-suited for detecting these patterns in large datasets. (A detailed description of the algorithm is found in Appendix A). By employing this model, we can pinpoint combinations of refactoring types that consistently appear together across multiple commits. This identification of frequent itemsets is crucial, as it provides insights into the common co-occurrences of refactoring types, enabling us to better understand the relationships and trends within our performance and non-performance-related refactoring datasets.

3. **Identify Refactoring Patterns Using Association Rule Mining:** Building upon the frequent itemsets identified in the previous step, our analysis proceeds to establish association rules among the refactoring types within transactions. Association rules reveal the likelihood of certain refactoring types co-occurring, providing valuable insights into the collaborative application of these refactoring types by developers. (A detailed description of the algorithm is found in Appendix A). To generate meaningful and reliable association rules, we employ an iterative process to determine appropriate threshold values: (1) Support threshold: We experiment with different minimum support values to control the frequency of itemsets considered in our analysis. (2) Confidence threshold: We adjust the minimum confidence level to ensure the strength of the associations we identify. Through this iterative process, we fine-tune the thresholds to optimize the trade-off between generality and specificity in the context of our datasets and analytic objectives. We concluded by setting our desired minimum thresholds for support and confidence at *0.03 and 0.3* respectively.

$$\text{Support}(X \Rightarrow Y) = \frac{\text{Frequency}(X \cup Y)}{\text{Total number of transactions}} \tag{3.12}$$

$$\text{Confidence}(X \Rightarrow Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)} \tag{3.13}$$

*Where $X$ and $Y$ are sets of refactoring types, and $X \Rightarrow Y$ represents the association rule "if $X$ occurs, then $Y$ is likely to occur".*

**Results:**

In this section, we interpret the results of the frequent itemsets (Table 3.9 and Table 3.10) and association rules (Table 3.11 and Table 3.12) for performance-related refactorings and

compare and contrast them to non-performance-related refactorings presented.

1. **Frequent Itemsets Results and Interpretation:** The frequent itemset analysis reveals important insights into the most common refactoring types applied in our dataset. Table 3.9 shows the 10 most frequent itemsets in our performance-related refactorings dataset with their corresponding support.

    - **Type-Related Refactorings:** Both datasets focus on type-related refactorings (e.g., "Change Variable Type"). These refactorings appear less frequently in performance-related contexts and are more evenly distributed among different type changes. The minor differences in frequency between non-performance and performance-related refactorings (as shown in Tables 3.9 and 3.10) suggest that developers apply a range of type modifications across both contexts, without a clear concentration in performance-related scenarios.
    - **Method-Related Refactorings:** “Extract Method" and “Change Return Type" appear with minimal difference in frequency between the performance-related and non-performance-related datasets. These observations suggest that method-level refactorings are applied in both contexts, with minimal differences in their frequency.
    - **Variable Extraction:** “Extract Variable" appears with a support of 14.55%. “Extract Variable" was also prominent in the non-performance-related refactorings (16.30%). The slightly lower frequency in performance-related refactorings might indicate that while variable extraction remains important, it is not as critical for addressing performance issues as it is for general code improvement.
    - **Naming Conventions:** The emphasis on renaming (e.g., “Rename Variable") is consistent across both datasets. This indicates that clarifying naming conventions remains important even when addressing performance issues, possibly to ensure code readability and maintainability alongside performance improvements.

2. **Association Rules Analysis Results and Interpretation:** The association rules analysis reveals patterns in how different refactoring types are combined in performance-related refactorings. Table 3.11 shows the 10 most association rules in our performance-related refactorings dataset with their corresponding support, confidence and lift.

    - **Strong Association between Parameter and Attribute Refactorings:** The strongest rule shows that "Rename Parameter" and "Change Attribute Type" strongly imply "Change Parameter Type" (confidence: 85.56%, lift: 7.42). This

Table 3.9 Top 10 Frequent Itemsets (Refactoring Types) in non-performance refactorings with their support

| Refactoring Types | Support |
|---|---|
| Change Variable Type | 0.1894 |
| Add Parameter | 0.1771 |
| Rename Method | 0.1710 |
| Extract Variable | 0.1630 |
| Rename Variable | 0.1587 |
| Extract Method | 0.1497 |
| Change Return Type | 0.1333 |
| Change Attribute Type | 0.1316 |
| Change Parameter Type | 0.1305 |
| Rename Parameter | 0.1230 |

Table 3.10 Top 10 Frequent Itemsets (Refactoring Types) in performance refactorings with their support

| Refactoring Types | Support |
|---|---|
| Change Variable Type | 0.1723 |
| Rename Variable | 0.1619 |
| Rename Method | 0.1606 |
| Add Parameter | 0.1606 |
| Extract Method | 0.1503 |
| Extract Variable | 0.1455 |
| Change Attribute Type | 0.1269 |
| Change Return Type | 0.1218 |
| Change Parameter Type | 0.1153 |
| Rename Parameter | 0.1149 |

Table 3.11 Top 10 Association Rules with Support, Confidence, and Lift in Performance Refactorings

| **LHS** | **RHS** | **Support** | **Confidence** | **Lift** |
|---|---|---|---|---|
| Change Attribute Type, Rename Parameter | Change Parameter Type | 0.033 | 0.856 | 7.421 |
| Rename Variable, Change Return Type | Change Variable Type | 0.032 | 0.796 | 4.619 |
| Change Attribute Type, Rename Method | Change Return Type | 0.032 | 0.716 | 5.878 |
| Change Return Type, Change Parameter Type | Change Attribute Type | 0.037 | 0.714 | 5.627 |
| Change Variable Type, Rename Method | Change Return Type | 0.031 | 0.692 | 5.686 |
| Change Return Type, Change Attribute Type | Change Parameter Type | 0.037 | 0.659 | 5.716 |
| Change Variable Type, Change Parameter Type | Change Return Type | 0.033 | 0.653 | 5.359 |
| Change Return Type, Change Parameter Type | Change Variable Type | 0.033 | 0.647 | 3.756 |
| Change Attribute Type, Change Parameter Type | Change Return Type | 0.037 | 0.620 | 5.096 |
| Change Variable Type, Change Attribute Type | Change Return Type | 0.033 | 0.611 | 5.019 |

**Note:** LHS represents the antecedent or the conditions, and RHS represents the consequent or the outcome. Support, Confidence, and Lift are metrics used to evaluate the strength and significance of the association rules.

Table 3.12 Top 10 Association Rules with Support, Confidence, and Lift in Non-Performance Refactorings

| **LHS** | **RHS** | **Support** | **Confidence** | **Lift** |
|---|---|---|---|---|
| Change Parameter Type, Rename Attribute | Change Attribute Type | 0.032 | 0.861 | 6.544 |
| Change Attribute Type, Rename Parameter | Change Parameter Type | 0.035 | 0.843 | 6.458 |
| Change Parameter Type, Rename Variable | Change Variable Type | 0.033 | 0.829 | 4.376 |
| Change Return Type, Rename Variable | Change Variable Type | 0.035 | 0.825 | 4.355 |
| Change Attribute Type, Rename Variable | Change Variable Type | 0.031 | 0.816 | 4.308 |
| Change Variable Type, Change Return Type, Change Attribute Type | Change Parameter Type | 0.034 | 0.805 | 6.170 |
| Change Return Type, Rename Parameter | Change Parameter Type | 0.031 | 0.774 | 5.929 |
| Change Parameter Type, Change Attribute Type, Change Variable Type | Change Return Type | 0.034 | 0.758 | 5.687 |
| Change Parameter Type, Change Return Type, Change Variable Type | Change Attribute Type | 0.034 | 0.755 | 5.741 |
| Change Parameter Type, Change Return Type, Change Attribute Type | Change Variable Type | 0.034 | 0.752 | 3.971 |

**Note:** LHS represents the antecedent or the conditions, and RHS represents the consequent or the outcome. Support, Confidence, and Lift are metrics used to evaluate the strength and significance of the association rules.

pattern is similar in both performance-related and non-performance-related datasets, with a higher lift value observed in performance-related refactorings. However, the difference in lift is not substantial enough to suggest that developers adopt a notably more focused approach to type optimization when addressing performance issues. The observed associations suggest that changes to parameter types are influenced by renaming parameters and changing attribute types in both contexts.

- **Variable Type Changes Associated with Return Type Changes:** “Change Return Type" and “Rename Variable" strongly imply “Change Variable Type" (confidence: 79.57%, lift: 4.62). This could be part of a strategy to ensure efficient data flow and type consistency throughout methods. Compared to non-performance-related refactorings, this association appears stronger in performance-related contexts, indicating a more focused approach to optimizing data types when addressing performance issues.
- **Method-Level Refactorings:** Several rules involve “Rename Method" as an antecedent: “Change Attribute Type" and “Rename Method" imply “Change Return Type" (confidence: 71.57%, lift: 5.88). “Change Variable Type" and “Rename Method" imply “Change Return Type" (confidence: 69.23%, lift: 5.69). These patterns suggest that method-level refactorings often involve comprehensive changes, including return types and associated variables or attributes.In contrast to non-performance-related refactorings, the presence of “Rename Method" in these high-confidence rules suggests that performance-related refactorings more often involve a holistic review and restructuring of methods.
- **Interplay between Return, Parameter, and Attribute Types:** Several rules show strong associations between changes to return types, parameter types, and attribute types: “Change Return Type" and “Change Parameter Type" imply “Change Attribute Type" (confidence: 71.43%, lift: 5.63). “Change Return Type" and “Change Attribute Type" imply “Change Parameter Type" (confidence: 65.89%, lift: 5.72). Developers seem to consider the entire type ecosystem of a class or method when making performance-related refactorings. While similar patterns exist in non-performance-related refactorings, the higher lift values in performance-related refactorings suggest that this holistic approach to type changes is more pronounced when addressing performance issues.
- **Consistency in Variable and Return Type Changes:** “Change Parameter Type" and “Change Variable Type" imply “Change Return Type" (confidence: 65.25%, lift: 5.36). This could be part of a holistic approach to optimizing within

methods. Compared to non-performance-related refactorings, this specific combination appears more frequently in performance-related contexts, indicating a more systematic approach to aligning parameter, variable, and return types for performance optimization.

*Our results reveal that while there are similarities in the refactoring types performed for general codebase improvement and addressing performance issues, there are subtle but important differences in their application. Performance-related refactorings seem to involve a more balanced and method-centric approach, with a comprehensive consideration of type optimizations across different code elements. These aspects should likely be considered as special requirements when recommending refactorings that are linked to performance issues.*

**RQ5. What are the different reasons for refactoring in performance issues?**

**Motivation:**

Refactoring code requires effort. Therefore, developers generally do not perform refactoring operations without a reason [12]. In this RQ, we aim to identify the different motivations and reasons behind refactoring activities performed on performance-related issues. It is our hope that our findings for this RQ can allow developers to make more informed decisions when selecting appropriate refactoring types when resolving performance-related issues.

**Approach:**

To answer this question, we sampled refactorings from the performance-related refactorings dataset taking into consideration only refactorings that contain at least one refactoring type from the top ten most prevalent refactoring types as shown in Table 3.7. Using a three-phase procedure, we manually analyze each refactoring commit by inspecting the diff of the commit, its corresponding commit message and related discussions on its project's GitHub issue tracker. We use our analysis to derive a common set of reasons that explain why refactoring was performed in the sampled commits and how various refactoring types were used. We then use our findings to produce a classification scheme that can be applied in future investigations.

*Classification Method*

We perform our classification using the three phases below:

1. **Phase 1: Performance-Related Refactorings Selection:** In the initial phase, we selected 50 random performance-related refactorings from our dataset. These refactorings were chosen with the aim of ensuring representation of at least one of the top 10 most prevalent refactoring types identified in each dataset, as outlined in Table 3.7. To achieve a balanced distribution across refactoring types, we selected 5 instances for each type. This selection process was designed to guarantee the uniqueness of each refactoring by ensuring that each instance had a distinct commit ID. Additionally, to avoid any bias towards specific projects, we sampled these refactorings across various subject systems, thereby ensuring a diverse and representative selection.
   To enhance the traceability and accessibility of each refactoring, we use information such as the FilePath, CommitId, issue title, issue URL, and commit URL obtained from both RefactoringMiner and the corresponding performance issues of each selected refactoring. By tagging each refactoring with these details, we provide a robust framework that allows for seamless navigation and quick access to related issue discussions and commit diffs on GitHub. This integration not only streamlines the process of tracing refactorings back to their origins but also supports a comprehensive examination of the changes and their impacts. Consequently, this method improves the ability to monitor, analyze, and understand the context and rationale behind each refactoring.

2. **Phase 2: Commit Inspection and Refactoring Classification:**

   - **Inspecting the commits:** Each of the authors manually inspected the commit diff, commit message, and discussion on the issue associated with each of the initial 50 selected refactorings. Each author inspected the commit diffs on GitHub with the help of a Google Chrome extension named Refactoring Aware Commit Review.[4] This extension uses RefactoringMiner to visually represent the diff with refactoring information. The extension adds extra information to the changes, like highlighting the type of change and its location. It can also identify if new code was added, moved, or changed and report all refactoring types used in a given refactoring.
   - **Identifying the reasons for refactoring and use of refactoring types:** By using this extension, we were able to manually identify the underlying reasons

[4] `https://chrome.google.com/webstore/detail/refactoring-aware-commit/lnloiaibmonmmpnfibfjjlfcddoppmgd`

behind each refactoring and the use of each refactoring type in a given refactoring-related commit by identifying the component of the code (such as class or method) where the resolution of the performance issues are performed.

- **Naming and classifying the reasons for refactoring and use of refactoring types:** Each author identified categories of *reasons and use* for each refactoring type in the selected refactorings. This classification process is carried out independently among the authors to guarantee the integrity and unbiased outcome of the classification for each author's inspection.
- **Reviewing and post-classification discussion:** Following this classification process, the authors met to conduct a post-classification discussion to unify the classifications made by each author. This discussion was a collaborative effort to ensure consistency and accuracy in categorizing the identified reasons and uses. During the discussion, the first author altered and updated the classification consensus accordingly. By engaging in this discussion, we reached a consensus and established a unified understanding of the reasons behind the refactorings.

3. **Phase 3: Finalising Classifications:** Using the unified understanding obtained from Phase 2, the first author manually inspected an additional 250 refactorings in our dataset. The author repeated the approach used in Phase 2 to identify the reasons behind these refactorings and the context in which the refactoring types were used. Any new categories were to be discussed with the other authors. However, no such instances arose. After, the authors conducted another post-classification discussion to validate and alter the outcome of the classification of the first author. In total, we classified a total of 300 performance-related refactorings.

**Results:**

A total of 300 commits corresponding to our selected performance-related refactorings were analysed. We then categorize the reasons behind these refactorings into ten categories. Each category represents a different aspect of the performance issue being addressed through refactoring. We also classify the context in which the various refactoring types were used into three categories. These categories are: *Fix-Related, Maintenance-Related and Test-Related*. Table 3.13 and Table 3.14 provide an overview of the distribution of the classified categories and the context in which the various refactoring types were used respectively.

Below we describe the categories of our classification and the prevalence of each category for

Table 3.13 Distribution of Classified Reasons for Performance-related Commits.

| Reasons | Count |
|---|---|
| Unrelated to Performance Improvement | 134 |
| Reliability and Stability Improvement | 91 |
| Execution Time Improvement | 31 |
| Memory Optimization | 15 |
| Scalability Improvement | 9 |
| I/O Optimization | 6 |
| Concurrency and Parallelism Enhancement | 5 |
| Caching Optimization | 4 |
| Network Performance Optimization | 3 |
| CPU Usage Optimization | 1 |
| Startup Time Reduction | 1 |

Table 3.14 Distribution of the Context in Which Refactoring Types are Used.

| Contexts of Use | Count |
|---|---|
| Maintenance-related | 188 |
| Fix-related | 93 |
| Test-related | 19 |

the top 10 refactoring types of performance-related refactorings, discussing the three most prevalent refactoring types for each category, other less prevalent refactoring types can be found in the replication package. Figure 3.2 provides a detailed overview of the classification of the performance-related commits.

1. **Unrelated to Performance Improvement (134/300):** This category encompasses refactorings that, while not directly targeting performance improvements, enhance overall software quality, and maintainability, and lay the foundation for future optimizations. The high prevalence of these refactorings in performance-related commits reveals that:

    - *Developers take a comprehensive approach to addressing performance issues, improving various aspects of the codebase simultaneously.* For example, in project *NewPipe* (commit: 1d42e45),[5] developers addressed a background player buffering issue while simultaneously improving code maintainability by standardizing how video details are passed to fragments. This example shows how the developers used the “Add Parameter" refactoring type in a performance-related commit. By adding parameters, they standardized how the `VideoDetailFragment` is opened, improving maintainability and flexibility.

[5]https://github.com/TeamNewPipe/NewPipe/commit/1d42e45d785b955f7f2dfc38c4da168cd7601e27

- *The bundling of non-performance-related refactorings with performance-focused changes suggests that developers seize the opportunity to improve overall code quality when working on performance issues.* For example, in project *nacos* (commit: d08eac6),[6] developers unified the implementation of HTTP requests, which enhances code maintainability and consistency. This example shows how the developers employed the “Change Return Type" refactoring type in a performance-related commit. By modifying the return type, they standardized the HTTP client implementation, making the codebase more consistent.
- *Well-organized code may leads to better performance in the long run, as it is easier to identify and implement optimizations in clean code.* For example, in project *graphhopper* (commit: 5cf826c),[7] developers changed a parameter type in the PrepareContractionHierarchies class to use dynamic flag assignment in flag encoders. This example shows how the developers used the “Change Parameter Type" refactoring type in a performance-related commit. By changing the parameter type, the developers improved code maintainability by using a different encoder type that may be more appropriate or provide additional functionality. The improved maintainability resulting from this change can make the code easier to understand, modify, and optimize in the future.

The most common refactoring types in this category show that developers solving performance issues have a strong focus on, improving the overall architecture and organization of the codebase (i.e., “Move Class" (25)), emphasizing code readability and maintainability (i.e., “Extract Variable" (23)), and place importance on improving code documentation and metadata (i.e., “Add Method Annotation" (17))).

> *The high occurrence of unrelated performance improvement in performance-related commits highlights the complex, intertwined nature of software quality and addressing performance issues. It demonstrates that effective performance issue resolutions often require a holistic approach to software development, where code structure, readability, and documentation are seen as integral parts of the resolution process.*

2. **Reliability and Stability Improvement (91/300):** This category encompasses refactorings that focus on enhancing the system’s dependability, error resistance, and overall robustness. With 91 out of 300 refactorings in this category, it highlights the critical importance of reliability in software systems. These refactorings contribute to

[6] https://github.com/alibaba/nacos/commit/d08eac68823fe7ff31b5d825173df9efc0048439
[7] https://github.com/graphhopper/graphhopper/commit/5cf826c8d089d7b4c9f1b1e319140d1b9451b82d

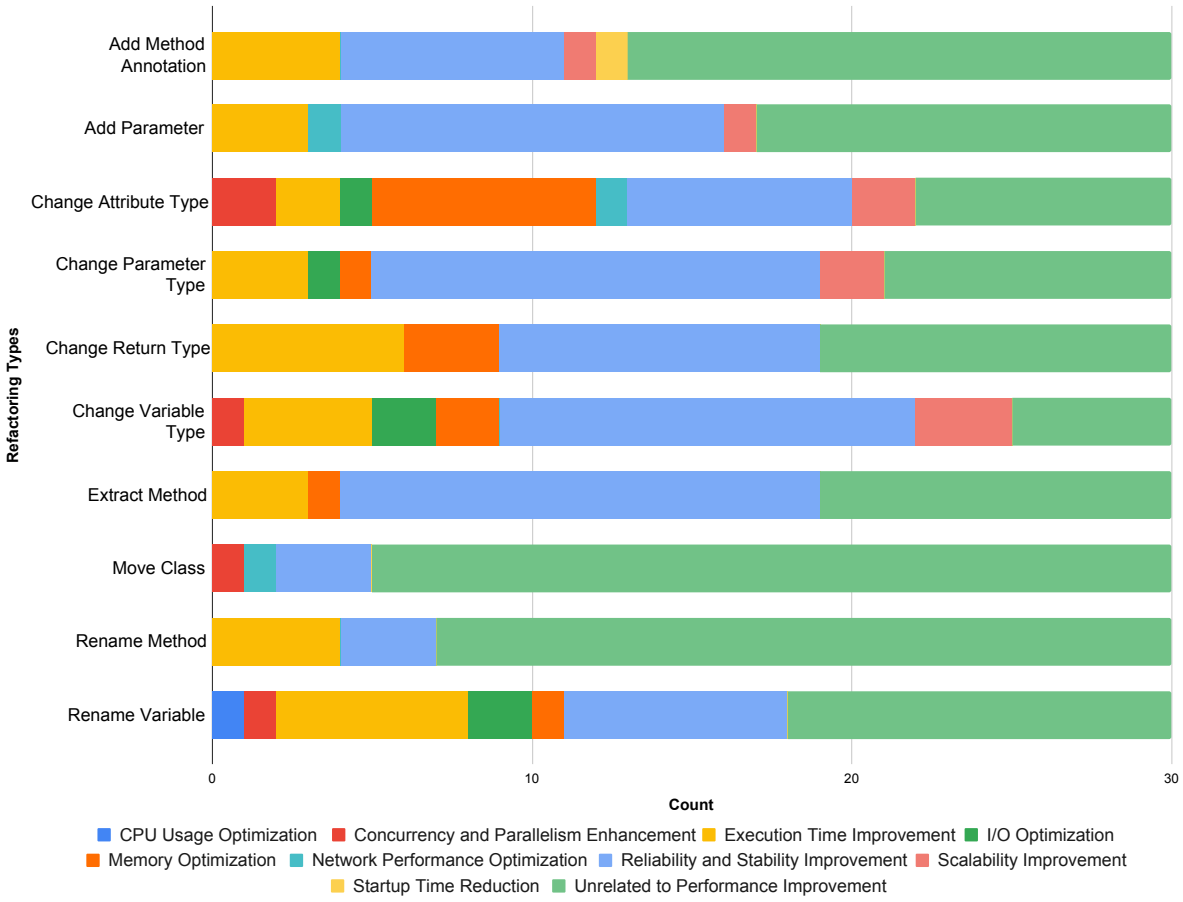


Figure 3.2 Prevalence of each category for the top 10 refactoring types of performance-related refactorings.

several key areas:

- *Extracting methods can centralize error handling logic, while changing parameters and variable types can introduce more robust error checking.* For example, in project *junit5* (commit: 3655040),[8] developers changed the parameter type in the `AssertTimeout` class to address an issue where timeouts were incorrectly reported instead of actual failing assertions. This example shows how the developers used the “Change Parameter Type" refactoring in a performance-related commit. By changing the parameter type, the developers improved the reliability and accuracy of the assertion mechanism. The refactoring contributes to more robust error handling by ensuring that the correct type of failure (timeout or assertion) is reported, leading to more reliable software.

[8]https://github.com/junit-team/junit5/commit/36550401836858a1e6987b06d4c15f3ed95c19d0

- *New or modified methods might include additional validation steps. Changed variable types could enforce stricter data constraints.* For instance, in project *skywalking* (commit: 2042709)[9], developers applied the "Change Attribute Type" refactoring type. They changed the attribute type of `clusterTransportSniffer` from `String` to `Boolean` in the `ElasticSearchClient` class. This refactoring enforces stricter data constraints preventing potential misuse or incorrect value assignments and thereby enhancing data integrity. This change was part of a feature aimed at improving reliability and stability.

The most prevalent refactoring types in this category show how developers solving performance issues approach reliability and stability improvements. Indeed, developers frequently isolate complex or error-prone logic into separate methods (i.e., "Extract Method" (15)); they strengthen type safety, introduce more specific or constrained types, and adapt method signatures to accommodate new error handling or logging ("Change Parameter Type" (14)); finally, they adopt robust data types to prevent unexpected behaviour ("Change Variable Type" (13)).

> *The substantial number of refactorings related to reliability and stability improvements (91/300) underscores the critical importance developers place on these aspects. It suggests that addressing performance issues often goes hand-in-hand with refactoring systems to be more reliable.*

3. **Execution Time Improvement (35/300):** This category focuses on refactorings that enhance system speed and efficiency, involving various optimization techniques. These refactoring types are often part of broader optimization strategies, including:

   - *Refactoring code to use more efficient algorithms or data structures might involve changing the core logic of methods, which could necessitate changes in return types or variable types.* For example, in project *lisa* (commit: fd9f321),[10] developers applied the "Change Variable Type" refactoring type. They changed the variable type from `unsorted sets` to `sorted sets (TreeSet)` in the `Automaton` class. By using a more efficient data structure, the overall execution time of algorithms working with these sets can be improved. This change was part of a feature request to refactor the project structure to improve execution time.
   - *Introducing caching mechanisms for frequently accessed data often requires method signature and return type changes to accommodate cache hits and misses.* For

[9] https://github.com/apache/skywalking/commit/2042709b029f303c0e35d678cbc8f271db0d8359
[10] https://github.com/lisa-analyzer/lisa/commit/fd9f321ab6f9ebab1cf43337431cd749a84a64f4

instance, in project *rstudio* (commit: 0ee548b),[11] developers applied the "Add Parameter" refactoring type. They added a new `String` label parameter to the `RAddinCommandPaletteEntry` constructor. While not a traditional caching mechanism, this approach serves a similar purpose by storing and reusing pre-computed data, potentially improving search performance in the command palette. This refactoring was implemented to address the issue of `slow rstudio_comm and_palette_search`, aimed at improving execution time.

- *Eliminating unnecessary calculations or deferring expensive operations can involve restructuring code and changing how variables are used and what types they hold.* For example, in project *DSpace* (commit: 2c9165a),[12] developers applied the "Change Variable Type" refactoring type in the `GroupServiceImpl` class to count members instead of retrieving and processing each one. This refactoring directly reduces computation by avoiding the overhead associated with handling large lists. This change reduces the amount of data processed and the number of operations performed, leading to improved performance and resource utilization. This refactoring was implemented to improve execution time with groups with many EPerson members.

The prevalence of specific refactoring types in this category provides interesting insights into how developers approach execution time improvements. Results suggest that developers optimize by adjusting method outputs (i.e., "Change Return Type" (6)). For instance, changing a method to return a more efficient data structure (e.g., from a List to a Set for faster lookups). While not directly affecting performance gains, developers also prioritize code readability and maintainability (i.e., "Rename Variable" (6)). Finally, developers also switch to more efficient data types (e.g., use primitive types instead of their object wrappers)(i.e., "Change Variable Type" (4)). The relatively low number of refactorings in this category might suggest that significant execution time improvements can require more substantial changes than simple refactorings (e.g., configuration changes or infrastructure upgrades), or that such optimizations are less frequently needed.

[11] https://github.com/rstudio/rstudio/commit/0ee548b4a80e5d5a638db04b13b369f4cad01b27
[12] https://github.com/DSpace/DSpace/commit/2c9165afb08126189ee3367347e7011f89227b7c

*While direct performance-related refactorings types like "Change Return Type" and "Change Variable Type" play a crucial role in execution time improvement, the presence of "Rename Variable" refactorings highlights the importance of code clarity in the optimization process. This balanced approach ensures that performance improvements are both effective and sustainable in the long term.*

4. **Memory Optimization (15/300):** This category focuses on refactorings that improve system memory management, crucial for performance in resource-constrained environments or systems handling large datasets. The distribution of refactoring types in this category provides valuable insights into how developers approach memory optimization:

   - *Developers refactor while reducing memory leaks, often requiring changes in how object references are managed.* For example, in project *killbill* (commit: 9c122ea)[13], developers applied the "Rename Variable" refactoring type in the **CacheControllerDispatcherProvider** class. This refactoring was part of addressing an issue with direct buffer leak detection when using Redis cache. By improving the naming and structure of the code, developers can better track and manage object references, making it easier to identify and fix memory leaks in the caching system.
   - *Developers refactor code to minimize object creation and destruction, particularly in performance-critical sections.* For instance, in project *FreeBuilder* (commit: 609ca1d)[14], developers applied the "Change Attribute Type" refactoring type in the `Metadata_Builder` class. They changed the attribute type from `ArrayList` to `ImmutableList`. `Immuta bleList` can be more memory-efficient as it does not need to allocate extra memory, and it can eliminate the need for synchronization. This refactoring minimizes unnecessary object creation and destruction, particularly beneficial in performance-critical sections of the code.

   The distribution of refactoring types in this category provides valuable insights into how developers approach memory optimization. Indeed, developers optimize memory usage by altering the data types of class attributes (i.e., "Change Attribute Type" (7)). They return more memory-efficient data structures from methods and implement lazy loading patterns (i.e., "Change Return Type" (3)). Finally, they also focus on optimizing local variable declarations for reduced memory footprint (i.e., "Change Variable Type" (2)).

[13]https://github.com/killbill/killbill/commit/9c122ea297d7ccd600f9b21e81874320c1203944
[14]https://github.com/inferred/FreeBuilder/commit/609ca1da5492f9ffd49aa1642cfc66ecdedfa6dc

*While the number of memory optimization refactorings is smaller compared to other categories, their impact can be substantial. The focus on changing attribute and return types highlights the importance of careful data structure selection and management in memory-efficient software design. These refactorings, although seemingly minor, can lead to improvements in system stability, performance, and overall resource utilization.*

5. **Scalability Improvement (9/300):** This category focuses on enhancing the system's ability to handle growth and increased load. The low number suggests that significant scalability improvements often require architectural changes rather than refactorings. The distribution of refactoring types in this category provides valuable insights into how developers approach scalability improvement:

   - *Changes in variable and attribute types might facilitate easier data partitioning or sharding, supporting horizontal scaling. Parameter type changes could allow for more flexible configuration, supporting both horizontal and vertical scaling strategies.* For example, in project *rdf4j* (commit: 16b10b8),[15] developers applied the "Change Parameter Type" refactoring type in the `LmdbLiteral` class to optimize how IDs are stored and managed. This refactoring aims to handle larger values more efficiently, which is crucial for horizontal scaling. This change was part of implementing an alternative embedded persistent backend, focusing on scalability improvement and fixing related issues in the system's capacity to handle growth, which is a key aspect of performance for large-scale or long-running systems.
   - *Modifying types can lead to more efficient resource usage.* For instance, in project Biomancy (commit: b3ad3b2),[16] developers applied the "Change Attribute Type" refactoring in the DigesterContainer class. This refactoring supports better scalability by managing resources more efficiently. By optimizing inventory management and expanding fuel options, the system can handle increased load more effectively.

   The most prevalent refactoring types in this category provide insights into how developers approach scalability improvements at the code level. Indeed, developers adopt more efficient data structures that perform better under increased load, improve parallelism, and allow lazy loading (i.e., "Change Variable Type"). They also attempt to

[15] https://github.com/eclipse-rdf4j/rdf4j/commit/16b10b8f7dba04f0ef7d69ab1679d1b987c6681b
[16] https://github.com/Elenterius/Biomancy/commit/b3ad3b2ee8b4218d319ece0af43af19fdee29a52

use more scalable data structures parameters and types (i.e., “Change Attribute Type" and “Change Parameter Type" (2)).

*While the number of scalability-focused refactorings is small, their potential impact is great. The focus on changing variable, attribute, and parameter types suggests that even minor adjustments in data handling and method signatures can contribute to improved scalability. These refactorings likely complement larger architectural decisions and infrastructure changes aimed at enhancing system scalability. The presence of these refactorings demonstrates developers' awareness of scalability concerns even at the code level, ensuring that the system is prepared to grow and adapt to increasing demands.*

The remaining 16 refactorings are shared between *I/O Optimization, Concurrency and Parallelism Enhancement, Network Performance Optimization, CPU Usage Optimization, and Startup Time Reduction.* These refactorings, while not numerous, present specialized opportunities for fixing performance issues. In **I/O optimization (6/300)**, developers focused on improving system interactions with input/output operations, as seen in the *NewPipe* project (commit 0cdfa6e)[17], where they optimized disk read/write operations by changing attribute types. **Concurrency and Parallelism Enhancements (5/300)** improved how systems handle multiple tasks simultaneously, exemplified by the *querydsl* project (commit 176a3d2)[18], where developers ensured thread safety by using `ThreadLocal <SimpleDateFormat>`. **Network Performance Optimization (3/300)** focused on enhancing system communication, as demonstrated in the *dataverse* project (commit 8a2eba6)[19], where the shift to an `EJB-managed SolrClientService` reduced network connections and improved system responsiveness. CPU usage optimization (1/300) and startup time reduction (1/300), while minimal in representation, addressed critical performance areas. In the *JSqlParser* project (commit 2aec1f6)[20], variable renaming was part of optimizing CPU-intensive parsing operations. Common refactoring types across these categories included Change Variable Type, Change Attribute Types, Move Class, and Add Parameter, all aimed at improving efficiency, thread safety, and code clarity.

[17] `https://github.com/TeamNewPipe/NewPipe/commit/0cdfa6e377e48002247ec515b7eb2a3353f5c007`
[18] `https://github.com/querydsl/querydsl/commit/176a3d215cd8adb855fb371e03e7d9aeee5118e2`
[19] `https://github.com/IQSS/dataverse/commit/8a2eba69f19d106f41105b99037b5cd9dce6eb95`
[20] `https://github.com/JSQLParser/JSqlParser/commit/2aec1f61ec4298ebc9c355af2ddcf86e55d7f8ba`

*While refactorings related to I/O Optimization, Concurrency and Parallelism Enhancement, Network Performance Optimization, CPU Usage Optimization, and Startup Time Reduction are few in number, they can greatly impact system performance. We believe that major optimizations in these areas may usually involve architectural changes or may be addressed through configuration or hardware improvements, which might not be captured as simple refactorings.*

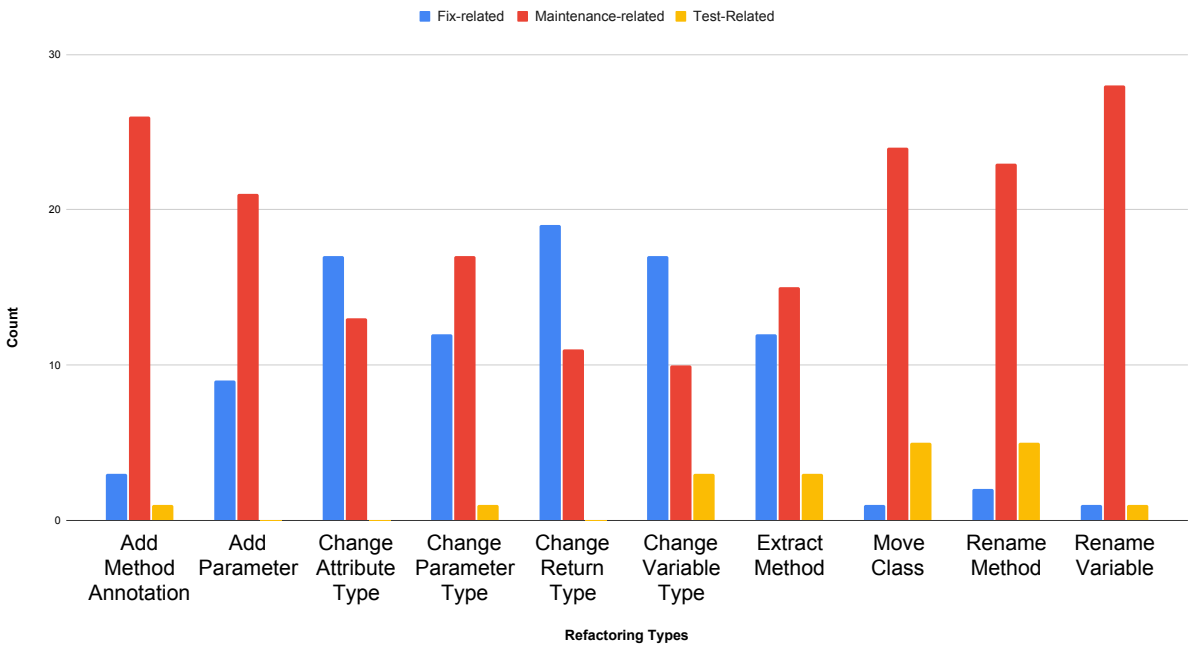


Figure 3.3 Prevalence of the uses of the top ten most prevalent refactoring types in relation to performance-related refactoring

While various refactoring types are indeed used for different purposes, they may also be applied in a variety of contexts, as illustrated in Figure 3.3. To better understand these contexts, we categorize the classification of the contexts in which various refactoring types were used in the subsequent paragraphs. This classification was done manually by analyzing each refactoring instance. We specifically looked at factors such as the nature of the changes, the surrounding code, and the commit messages to decide whether a refactoring was maintenance-related, fix-related, or test-related. (1) Maintenance-related refactorings typically involved code cleanup, restructuring, or optimization without altering functionality. (2) Fix-related refactorings were identified by their focus on resolving bugs or issues, often indicated by the context in which they were applied. (3) Test-related refactorings were those that improved, added, or modified testing code to ensure better coverage or reliability.

1. **Maintenance-Related (188/300):** This category encompasses refactoring types that were not directly used to resolve reported issues but served as supplementary techniques during the resolution process. With 188 out of 300 refactoring types falling into this category, it represents the majority of the performance-related refactorings observed, highlighting the importance of ongoing code maintenance when addressing performance issues.

   The distribution of refactoring types in this category provides insights into how developers approach code maintenance alongside performance improvements:

   - "Rename Variable" (28): While not directly improving performance, aids in understanding code, which can help identify and optimize performance bottlenecks, reduce cognitive load, and improve maintainability. For example, in the *JSqlParser* project (commit: 2aec1f6),[21] variables were renamed in the `TestUtils` class to address a CPU burn issue when parsing PostgreSQL queries, likely as part of a larger refactoring effort to identify hotspots in the code.
   - "Add Method Annotation" (26): Highlights the importance of improving code documentation and metadata, which can enhance static analysis, IDE support, and testing practices, potentially catching performance issues early. For example, in the *Payara* project (commit: 7b9efc2),[22] method annotations were added to improve caching mechanisms as part of an upgrade to `Hazelcast 3.5` for a `JCache` fix, enhancing metadata for analysis and tooling.
   - "Move Class (24): Reflects efforts to improve code organization and structure. For example, in the *WSO2 product-apim* project (commit: 2f35cdd),[23] classes were moved within the admin-clients module as part of addressing concurrency and parallelism issues, ultimately aiding in isolating performance-critical sections.

   *The high number of maintenance-related refactorings demonstrates that addressing performance issues is not viewed in isolation but as part of a broader effort to improve and maintain code quality. This approach likely leads to more sustainable performance improvements and a codebase that is easier to optimize and maintain over time. It underscores the importance of code maintenance as an integral part of the performance issues resolution process, rather than addressing performance issues as a separate concern.*

[21] https://github.com/JSQLParser/JSqlParser/commit/2aec1f61ec4298ebc9c355af2ddcf86e55d7f8ba
[22] https://github.com/payara/Payara/commit/7b9efc2ee537d4417c9a59493ec79b3a66169cf1
[23] https://github.com/wso2/product-apim/commit/2f35cddd1f8159a375c9c307e7648aaa87f81da7

2. **Fix-Related (93/300):** This category encompasses refactoring types that were specifically used to resolve reported bugs or issues, playing a vital role in the resolution process. With 93 out of 300 refactorings falling into this category, it indicates that a significant portion of performance-related refactorings are directly applied to fix identified performance issues. The distribution of refactoring types in this fix-related category is particularly informative:

    - "Change Return Type" (19): This refactoring can impact performance by reducing data transfer overhead or improving data handling. For example, in the *quarkus* project (commit: 3a12609),[24] developers changed the return type in the Cache integration module to improve performance by reducing unnecessary data copying and processing overhead, leading to more efficient caching operations.
    - "Change Attribute Type" and "Change Variable Type" (17): These refactoring types are frequently used to address performance issues by modifying data representations at both the class and method levels, indicating that some performance problems stem from how data is stored and manipulated. For example, in the *querydsl* project (commit: 176a3d2),[25] the attribute type was changed to `ThreadLocal<SimpleDateFormat>` to improve concurrency and thread-safety in date parsing operations. This refactoring reduced contention and synchronization overhead, significantly improving performance and reducing concurrency-related bugs.

    *The prevalence of type-changing refactorings in fix-related scenarios reveals several key points about how developers approach performance problem-solving: (1) The focus on changing types (return, attribute, variable) suggests that developers often find performance gains by optimizing how data is represented and passed around in the system. (2) These refactorings represent relatively small, targeted changes rather than large-scale restructuring, indicating a preference for precise, focused fixes to performance issues. (3) The frequency of return type changes suggests that developers often find performance improvements by adjusting the interfaces between components, not just internal implementations. (4) The use of these refactoring types requires careful consideration to maintain correct functionality while improving performance, highlighting the delicate balance developers must strike.*

[24] https://github.com/quarkusio/quarkus/commit/3a1260954b51e976834306c1b4163c527c88f877
[25] https://github.com/querydsl/querydsl/commit/176a3d215cd8adb855fb371e03e7d9aeee5118e2

3. **Test-Related (19/300):** This category encompasses refactoring types that were used specifically in the test components of the codebase, rather than directly resolving reported bugs or issues. With 19 out of 300 refactorings falling into this category, it represents a small but significant portion of the performance-related changes, highlighting the importance of maintaining and improving test suites alongside production code. The distribution of refactoring types in this category provides insights into how developers approach test code maintenance in addressing performance issues:

    - "Rename Method", "Move Class (5) and "Extract Method, "Change Variable Type" (3): These refactorings are common in test-related changes, reflecting a focus on clarity, organization, and alignment of test code with the main codebase. For instance, in the *vertx-ignite* project (commit: b5a014e),[26] test classes related to `Ignite` cache expiration were moved to a more organized package structure. By renaming methods and moving classes, developers ensured that the test suite mirrored the reorganized production codebase, aiding in the identification and isolation of performance-related issues during testing. Also, in the *quarkus* project (commit: f1bed8a),[27] methods were extracted in cache integration tests to modularize scenarios related to Redis caching, enhancing reusability. Additionally, variable types were updated in tests to match changes in the production code's cache handling, ensuring accurate simulation of real-world scenarios for detecting performance regressions and validating optimizations.

*Our findings underscore the importance of maintaining and evolving the test suite alongside performance improvements in the main codebase. The (19/300) instances of test-related refactorings highlights the symbiotic relationship between production code and test suite evolution, emphasizing that effective performance management involves not only changing production code but also ensuring those changes are reflected and verifiable through a well-maintained test suite.*

### 3.4 Conclusion

In this chapter, we target 31,614 open-source Java projects for an empirical study where we investigate the prevalence, use, and reasons behind refactoring activities in performance issues. After filtering, we collect performance-related issues, commits and pull requests for 255 of these projects using the GitHub API. Using RefactoringMiner[26, 27], we mine our

[26]https://github.com/vert-x3/vertx-ignite/commit/b5a014e51204f0677653814aab08070f1a964a2c
[27]https://github.com/quarkusio/quarkus/commit/f1bed8a64e4aa3d0542b0b02318573e7873aacb4

subject systems and extract 19,646 performance refactorings and 9,860,235 non-performance refactorings with which we construct our refactoring datasets. We perform multiple statistical analyses to determine the prevalence and frequency of refactoring types in performance and non-performance refactorings. We also manually analyse our datasets to derive a classification scheme to classify the motivations behind the refactoring of performance issues and how various refactoring types are used. We use the *Association Rule Mining* approach, specifically the *Apriori algorithm* [28] to identify patterns of co-applied refactoring types applied in both performance-related and general refactorings (ie; a combination of performance-related and non-performance-related refactorings). We extract three overall lessons from our study:

- **Performance-related software refactoring activities have distinct characteristics.** For example, our results show that performance issues are primarily addressed by modifying class structures and method configurations. Indeed, the “Change Variable Type" refactoring is the most impactful across our studied projects, with an average frequency of 7.09 and 1807 occurrences across projects. Also, method-level refactorings like “Extract Method" and “Add Parameter" are preferred over complex class-level changes, indicating that developers favour targeted, low-risk adjustments when addressing performance issues. These specific characteristics hint at the need for more targeted refactoring recommendations for developers undertaking performance-related activities.

- **The motivation for, and activities performed during, performance-related software refactoring activities differ from those of non-performance-related refactorings.** Our manual classification of 300 performance refactorings from 113 projects revealed 11 reasons for refactoring in performance-related issues. Notably, we observe a high occurrence of refactorings unrelated to performance improvement, indicating that performance-related commits often address broader software quality concerns. Furthermore, we identified three contexts for performance-related refactorings: Fix-Related, Maintenance-Related, and Test-Related. These findings can be used to provide examples and guides for both researchers and developers looking to improve how software projects balance various aspects of software quality when dealing with performance issues.

- **The patterns of co-applied refactorings types used in performance-related refactorings differ from those of non-performance-related refactorings.** In performance-related refactorings, we find stronger associations between parameter and attribute modifications. The data suggests that developers adopt a more thorough ap-

proach to type optimization when addressing performance issues. Method-level refactorings in performance contexts often involve comprehensive changes, including return types and associated variables or attributes. These findings highlight the complex, interconnected nature of refactoring decisions in performance issue resolution, offering valuable insights into developers' strategies for improving software performance.

Overall, our results, through the identified refactoring patterns and our classifications, show how and why performance-related refactorings are useful in addressing performance issues, and merit special consideration to aid developers who seek to improve the performance of their systems.

# CHAPTER 4 UNDERSTANDING WHY REFACTORING IMPACT SOFTWARE PERFORMANCE

## 4.1 Introduction

Refactoring is a fundamental practice in software engineering aimed at improving code maintainability, readability, and quality without altering external behavior [8, 11, 14]. However, while refactoring is generally intended to enhance software structure, it can also influence execution time, sometimes yielding performance improvements and at other times causing regressions [5, 20]. Despite extensive research on refactoring's impact on software quality attributes [21, 40], limited studies systematically examine its direct effects on performance.

This study investigates the relationship between refactoring and execution time by analyzing 15 open-source Java projects. We employ RefactoringMiner, JPerfEvo, and the Java Microbenchmark Harness (JMH) to track 1,559 method-level changes and 11,317 refactorings, systematically measuring their impact on execution time. Our analysis highlights that method extraction and variable inlining are the most frequently associated with execution-time variations, with some refactorings improving performance while others introduce regressions. Additionally, we identify key contextual factors such as method invocation frequency, code structure, and test coverage that influence the magnitude and direction of performance changes.

To structure our investigation, we pose the following research questions:

- **RQ1:** Which specific refactoring types coincide with performance changes, and to what extent?
- **RQ2:** How much does performance change when refactorings coincide with performance changes?
- **RQ3:** What happens in refactorings when refactorings coincide with a performance change?

By addressing these questions, this study provides empirical insights into the performance implications of refactoring, helping developers make informed decisions when refactoring performance-sensitive code. Our findings contribute to the broader discussion on balancing maintainability and runtime efficiency, offering a replicable methodology for evaluating refactoring-induced performance variations.

## 4.2 Study Design

In this section, we present the comprehensive design of our study aimed at examining how refactoring operations affect software performance. Software performance is a complex attribute that can be evaluated through various metrics, including but not limited to response time, resource utilization, throughput, and scalability. Recognizing the multifaceted nature of performance measurement, our study focuses on execution time as the primary metric of interest.

*Execution time* is defined in this context as the duration required for a particular segment of code to execute from start to finish [85][86]. This measurement is conducted under controlled conditions where there is no concurrency or resource contention from other software processes running on the same platform. By isolating execution time, we eliminate external factors that could skew the results, thereby ensuring that any observed changes in performance can be directly attributed to the refactoring operations performed. To achieve this, we designed a series of steps that involve repeated benchmark executions in a controlled environment, systematic warm-up periods to stabilize JVM performance, statistical outlier detection and removal, and standardized hardware configurations across all test runs. This rigorous approach helps minimize the impact of variations and ensures reliable performance measurements.

### Benchmark Pipeline Tool

Collecting accurate execution-time performance data in evolving software systems presents significant challenges, particularly when measuring the impact of code modifications. While various profiling and performance measurement tools exist, the current research landscape lacks a unified framework that can seamlessly manage the entire process. Researchers and practitioners must often cobble together disparate tools and scripts to handle different aspects of the analysis pipeline: version control integration for tracking code changes (like Git[1] or SVN[2]), build system configuration for different versions (through Maven[3] or Gradle[4]), benchmark suite management (using JMH[5] or Caliper[6]), performance data collection (via

[1] `https://git-scm.com/`
[2] `https://subversion.apache.org/`
[3] `https://maven.apache.org/`
[4] `https://gradle.org/`
[5] `https://github.com/openjdk/jmh`
[6] `https://github.com/google/caliper`

JProfiler[7] or YourKit[8]), and results aggregation (through R[9] or Python[10] scripts). This fragmentation not only increases the complexity of performance studies but also introduces potential inconsistencies between different stages of the analysis process, from identifying relevant code changes to executing controlled performance benchmarks and instrumentation.

To address these challenges, we use **JPerfEvo**[1], a comprehensive benchmarking and instrumentation pipeline designed specifically for Java systems. JPerfEvo employs direct Java Virtual Machine (JVM) attachment for method instrumentation during benchmark execution, achieving precise performance data collection. The tool's architecture provides an integrated workflow that orchestrates the entire performance analysis process: from commit collection and filtering, through code change extraction and analysis, to automated benchmark construction and execution. By unifying these steps into a single, coherent pipeline, JPerfEvo provides a robust and practical environment for researchers and practitioners to systematically study how code changes influence execution-time performance. Using JPerfEvo, our study follows these steps:

1. *Commit History Analysis and Filtering:* We use a multi-stage filtering pipeline that processes the complete commit history of a project. This excludes merge commits to prevent duplicate analysis and eliminate noise in performance measurements. We process only commits containing modifications to Java source files (`.java` files) and exclude changes isolated to test files, ensuring focus on the project's production codebase. This filtering process yields a refined set of commits, each containing valid Java method modifications that serve as input for subsequent stages of the analysis pipeline.

2. *Code Change Extraction:* We analyze and extract detailed method-level changes from Java source code, eliminating commits that do not contain meaningful method-level modifications. This enables the extraction of essential method information including method names, identifiers, return types, access types, and full declaring class identifiers [87].

3. *Benchmark Build Process:* We build JMH benchmarks across different project configurations. JPerfEvo supports two common benchmark organization patterns: benchmarks defined as separate modules in `pom.xml` (built using `mvn -pl [jmhModule] -am clean package`), and benchmarks integrated within the main project structure (requiring `mvn clean install` followed by `mvn clean package` in the benchmark directory). Both

[7] `https://www.ej-technologies.com/products/jprofiler/overview.html`
[8] `https://www.yourkit.com/`
[9] `https://www.r-project.org/`
[10] `https://www.python.org/`

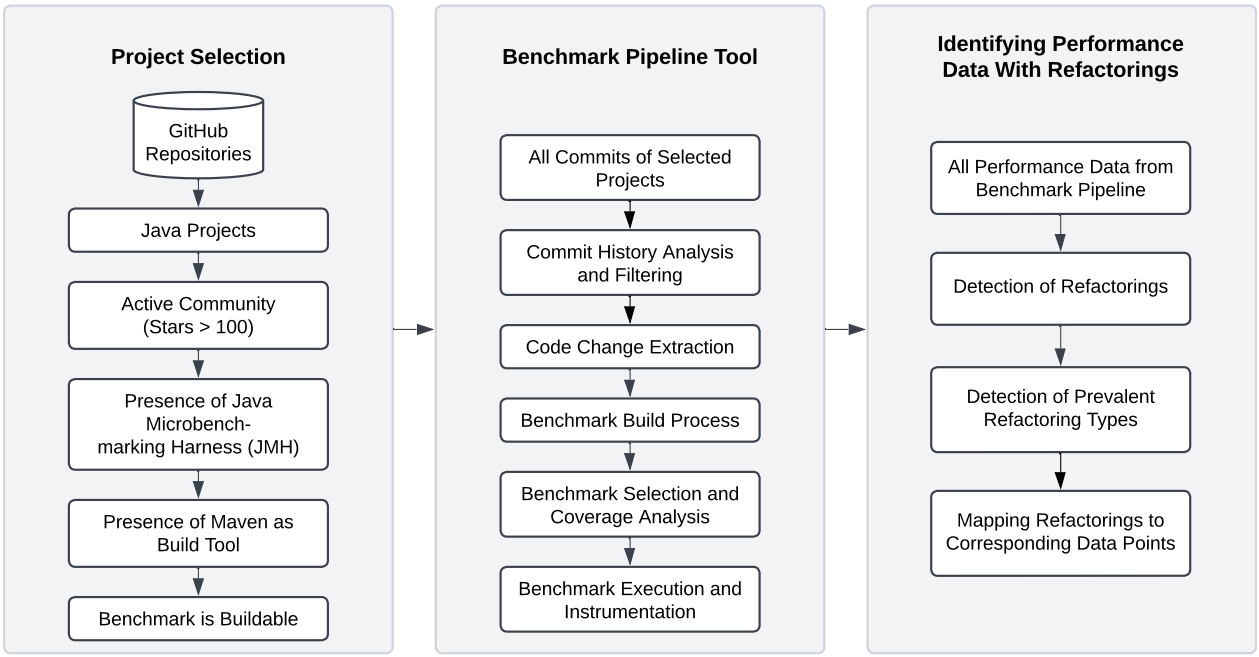


Figure 4.1 Study Design Overview

approaches yield executable JMH benchmark `.jar` files. JPerfEvo implements automated build failure recovery by adjusting build parameters and environment settings, such as Java version modifications or using project-specific Maven wrappers. Commits that persistently fail the build process despite these recovery attempts are automatically excluded from the analysis pipeline.

4. *Benchmark Selection and Coverage Analysis:* We optimize benchmark execution by selecting relevant microbenchmarks for changed methods. This selects the minimal benchmark set that achieves maximum coverage. Commits without benchmark coverage for their modified methods are excluded from analysis, acknowledging the typically lower coverage of performance benchmarks compared to unit tests.

5. *Benchmark Execution and Instrumentation:* We execute selected benchmarks with instrumentation to collect method-level performance data. For each commit and its predecessor, benchmarks are run with user-defined settings (e.g., number of forks, iterations, and warm-ups) and minimal overhead (less than 2%). By applying identical execution conditions across commit pairs, we ensure that observed performance changes can be reliably attributed to code changes rather than environmental factors. Since unexpected internal or external interruptions may affect the benchmarking process, we

rely on **JPerfEvo**[11] to remove outliers from the distribution using its Interquartile Range (IQR) technique.

### Experimental Setup

We executed JPerfEvo on a dedicated workstation to process our selected projects. The workstation was a Google Cloud Platform Compute Engine provisioned with virtual machines, each equipped with an 8-core CPU, 16 GB of memory, and 250 GB of SSD storage, providing consistent resources for build and benchmark operations. To maintain experimental integrity and minimize external interference, we ensured the workstation remained isolated from other tasks during the benchmarking process. This controlled environment enabled reliable performance measurements across all analyzed projects.

### Selection of Projects

Our project selection process began by identifying Java projects from prior works [20] that met established criteria for performance benchmark analysis. We focused on this specific set of projects to enable direct comparisons with previous findings while acknowledging the significant computational demands of our methodology each project requires approximately 40 or more compute hours for comprehensive benchmark execution and analysis. This imposed a limit to our study due to resource constraints. Thus, we focused on projects that incorporated Java Microbenchmarking Harness (JMH), used Maven as the dependency manager, and maintained a minimum of 100 GitHub stars indicating active community engagement. Additionally, the projects needed to provide clear benchmark build and execution instructions in their documentation and contain automatically executable benchmarks. From an initial set of 31 projects meeting these criteria, we manually conducted a detailed verification process to ensure compatibility with our pipeline. We analyzed each project's build configuration, focusing on projects where benchmarks could be built using the standard Maven command format `mvn -pl [jmhModule] -am clean package`. Through a comprehensive review of GitHub repositories, official documentation, and build configurations, we identified 15 projects (see Table 4.1) that fully aligned with our benchmark pipeline requirements and could support reliable, automated performance analysis.

### Benchmark Execution and Performance Data Collection

We ran each of the 15 projects through JPerfEvo to systematically analyze their performance characteristics. JPerfEvo's comprehensive pipeline begins with Git history analysis,

[11] https://github.com/kavehshahedi/java-instrumentation-buddy

performing multi-stage filtering that includes both basic commit validation and advanced quality checks. The pipeline verifies build status through GitHub's CI pipeline API endpoints, ensures the presence of essential build configuration files, and identifies Java version requirements for each commit, defaulting to Java 8 when not specified.

Given computational constraints, the pipeline implements a systematic sampling strategy to select representative commits from each project. Rather than random sampling, which could yield unbuildable commits, we employ a statistical sampling technique that calculates a sampling interval $k$ based on the total number of commits $N$. This approach ensures balanced representation across each project's development timeline while prioritizing commits with more method-level changes, which were likely to yield more performance data.

For each selected commit, JPerfEvo extracted detailed method-level changes, capturing method signatures and structural modifications while normalizing non-performance-impacting changes. The pipeline then built the project's benchmarks according to their specific Maven configurations, automatically handling build-time dependencies and environment requirements. JPerfEvo identified relevant benchmarks through coverage analysis, selecting the optimal set of microbenchmarks that exercised the modified methods. These selected benchmarks were executed for both the current commit and its predecessor, with standardized settings of 3 forks and 5 iterations, providing 15 total iterations for statistical reliability. During execution, method-level performance data were collected with minimal overhead, ensuring that observed performance changes could be accurately attributed to code changes.

This systematic approach through the JPerfEvo pipeline enabled us to collect comprehensive method-level performance data for 1559 data points (ie; method signatures) in 739 commits across all analyzed projects. This process took approximately 80 machine days.

### Identifying Refactoring-Related Performance Data Points

We define *Refactoring-Related Performance Data Points* as data points in our performance data which have refactoring operations performed within their commits.

### Detection of Refactorings

To identify refactoring-related performance data points, we use *RefactoringMiner* **(author?)** [26, 27], a state-of-the-art tool specifically designed for detecting refactoring activities in Java projects. For each data point in our performance dataset, the corresponding commit ID was passed to RefactoringMiner. The tool then performed a detailed analysis of the abstract syntax trees (ASTs) of the codebase associated with each commit to identify structural mod-

ifications that are indicative of refactoring operations.

Through this process, RefactoringMiner detected a diverse and extensive set of refactorings, resulting in a comprehensive dataset comprising 11,317 refactorings spanning 90 unique refactoring types. These refactorings included common types such as method extractions, renamings, and class reorganizations.

In addition to identifying refactoring-related performance data points, we used RefactoringMiner to extract all refactorings from the complete commit history of each project in our dataset. This broader extraction allowed us to establish a baseline by comparing the total number of refactorings across all commits to the subset of refactorings specifically associated with performance changes. This comparison aims to provide critical insights into the prevalence of refactoring-related performance data points in the context of overall code evolution. We collected a total of 291,887 refactorings across our 15 studied projects from this process.

**Detection of Prevalent Refactoring Types**

We hypothesize that not all refactoring types equally influence performance. Thus, we focus on identifying those that meet two key criteria: (1) the potential to impact performance at the method level, and (2) prevalence across the studied projects. Certain refactoring types were excluded from consideration due to their limited relevance to performance impact. These included:

- **Rename-related refactorings:** *Rename Method*, *Rename Class*, *Rename Variable*, *Rename Parameter*, and *Rename Attribute*.
- **Package-related refactorings:** *Change Package* and *Move Class*.

To identify the most prevalent refactoring types, we aggregated the remaining refactorings and ranked them using two metrics:

- **Frequency of occurrences:** The total number of times each refactoring type was observed across all projects.
- **Project coverage:** The proportion of projects in which a given refactoring type appeared.

We then identified the union of the top 10 ranked refactoring types based on these metrics, resulting in a subset of 12 unique prevalent refactoring types. Additionally, we included

*Inline Method* in this subset, despite it falling outside the top 10 rankings, as prior studies [20] highlighted its potential relevance to performance.

Using this refined set of prevalent refactoring types, we filtered the initial dataset to create the *selected refactoring dataset*, comprising 5,379 refactorings. This curated dataset served as the foundation for subsequent analyses.

### Mapping Refactoring Types to their Corresponding Data Points

Each refactoring-related performance data point commit contains multiple refactoring types and methods. To identify the specific refactoring type associated with the method signature in a refactoring-related performance data points commit, we conducted a manual mapping process. RefactoringMiner provides detailed metadata for each detected refactoring, including method signatures and commit IDs. These metadata were systematically compared against those in the refactoring-related performance data points to establish accurate associations. This mapping allowed us to associate our prevalent refactoring types (See 4.2) with corresponding refactoring-related performance data points. The outcome of this process was a refined dataset containing 191 refactoring-related performance data points spanning 12 out of 13 of the prevalent refactoring types where at least one of the prevalent refactoring types was applied (see Table 4.3). This dataset enables a focused analysis of the impact of these prevalent refactoring types on software performance.
While automation was considered, the complex nature of matching refactorings to performance changes required manual validation to ensure accuracy. Automated matching could lead to false positives due to the challenges in precisely attributing performance changes to specific refactoring operations, especially in commits containing multiple code changes. Furthermore, understanding the performance impact required careful comparison of code before and after the refactoring, often across multiple classes and files, to trace how the structural changes affected execution paths. This deep analysis of code changes and their relationships across the project's codebase necessitated human judgment and made full automation impractical while maintaining the reliability of our findings.

## 4.3 Study Results

In this section, we present the motivation, approach, and the results of each of our three research questions:

Table 4.1 Project performance metrics and method changes analysis.

| **Project** | **Total Commits** | **Stars** | **PC** | **PCwR** | **PM** | **PMwR** | $\mathbf{R_{PerfC}}$ | $\mathbf{RwPRT_{PerfC}}$ |
|---|---|---|---|---|---|---|---|---|
| apm-agent-java | 3,066 | 567 | 86 | 19 | 176 | 27 | 1931 | 889 |
| chronicle-core | 3,911 | 576 | 2 | - | 3 | - | 193 | 48 |
| client_java | 866 | 2,155 | 9 | 2 | 11 | 3 | 104 | 44 |
| fastjson2 | 4,372 | 3,769 | 220 | 30 | 615 | 34 | 1545 | 1055 |
| feign | 2,063 | 9,430 | 54 | 15 | 114 | 17 | 539 | 253 |
| hdrhistogram | 779 | 2,173 | 9 | - | 15 | - | 92 | 35 |
| jctools | 1,043 | 3,572 | 26 | 7 | 52 | 14 | 559 | 278 |
| jdbi | 5,709 | 1,984 | 90 | 17 | 136 | 20 | 962 | 328 |
| jersey | 798 | 692 | 23 | 5 | 37 | 5 | 731 | 312 |
| jetty.project | 30,160 | 3,858 | 56 | 12 | 124 | 16 | 1647 | 671 |
| netty | 11,604 | 34,850 | 57 | 14 | 97 | 21 | 470 | 250 |
| objenesis | 1,049 | 604 | 12 | 1 | 14 | 3 | 68 | 21 |
| protostuff | 1,603 | 2,050 | 4 | - | 4 | - | 143 | 136 |
| simpleflatmapper | 3,433 | 440 | 45 | 15 | 68 | 20 | 829 | 409 |
| zipkin | 2,955 | 16,982 | 46 | 11 | 93 | 11 | 1504 | 650 |
| **Total** | **73,411** | **-** | **739** | **148** | **1,559** | **191** | **11,317** | **5,379** |

**PC**= Performance Commits, **PCwR**= Performance Commits with Refactorings, **PM:**= Performance Methods, **PMwR:**= Performance Methods with Refactorings, $\mathbf{R_{PerfC}}$**:**= Refactorings extracted from Performance Commits, $\mathbf{RwPRT_{PerfC}}$**:**= Refactorings with Prevalent Refactoring Types extracted from Performance Commits

**RQ1 Which specific refactoring types coincide with performance changes, and to what extent?**

**Motivation:**

Refactoring is traditionally positioned as an activity that enhances code quality, maintainability, and readability while leaving the system's observable behaviour unchanged [6, 44]. Although performance often is not the primary goal of refactoring, the interplay between structural modifications and runtime behaviour cannot be overlooked. Certain refactorings, such as extracting a method or changing variable types, may alter the complexity of the code's execution path, resource usage patterns, or compiler optimizations. Over time, these subtle shifts can manifest as measurable performance changes either beneficial or detrimental. We define *performance changes* as any modification in a system's code, configuration, or environment that affects its execution time. This change can be either an improvement or a degradation

Bringing together our two focal points; (1) how frequently performance changes occur alongside refactorings, and (2) which specific refactoring types are most commonly associated with these performance changes provides a more comprehensive understanding of the impact of refactoring on performance. Such insights can help guide practitioners in anticipating potential performance implications when refactoring and can inform researchers who wish to examine the root causes and broader impacts of these changes. Ultimately, knowing the prevalence, but also the specific types of refactorings that appear in tandem with performance changes can help software teams prioritize their efforts and refine best practices for code evolution.

**Approach:**

We address this research question through a two-step approach:

1. *Assessing the prevalence of refactorings in our performance dataset:* We initiate our analysis by examining our performance dataset (see Section 4.2). To establish a baseline, we quantify our performance dataset by counting the number of *unique projects*, *unique commits*, and *method changes*, that experience performance changes, providing a comprehensive measure of performance variability. Subsequently, we filtered our performance dataset to identify only refactoring-related performance data points (see 4.2), creating a refined subset of our performance dataset. For this subset, we recalculated the metrics (*unique projects*, *unique commits*, and *method changes*) to allow for

a direct comparison. By analyzing the size and coverage of this subset with the entire performance dataset, we were able to quantify the relative frequency and scope of performance changes that co-occur with refactorings. This comparison provides valuable insights into the prevalence of refactorings in performance changes

2. *Identifying the significance of the refactoring types co-occurring with performance changes:* We analyze our performance dataset to identify and count refactoring types, determining which are most commonly associate to performance changes. We also analyze the distribution of refactoring types across each project's commit history by extracting and counting all refactorings. Comparing the total refactorings to the ones in the performance dataset helps establish a baseline for their relative significance. Furthermore, we analyze the relationship between refactoring types and performance changes by computing odds ratios [88] (see Appendix A), comparing their occurrence in the performance dataset against the refactorings in the general codebase (ie; refactoring dataset). This calculation considers both the presence and absence of each refactoring type in both datasets. An odds ratio greater than 1 suggests a stronger association with performance changes, while a value less than 1 indicates a weaker or inverse relationship.

**Results:**

Our analysis reveal a meaningful overlap between refactoring activities and performance changes:

1. *Prevalence of Refactorings in Performance Data:* **The results of our analysis (see Table 4.2) reveal that 80% of the projects in our study exhibit at least one instance where refactorings coincided with performance changes. Also, 20% of commits and 12.6% of methods of our performance dataset are associated with refactoring operations**. These findings suggest that refactorings and

Table 4.2 Summary of Performance Dataset and Refactorings Counts.

| | **Projects** | **Commits** | **Methods** |
|---|---|---|---|
| Performance Data | 15.0 | 739.0 | 1559.0 |
| Refactoring-Related Performance Data Points | 12.0 | 148.0 | 196.0 |
| % in Performance Data | 80.0 | 20.0 | 12.6 |

performance change co-occurrences are relatively common, reflecting a consistent pattern across software projects. This prevalence indicates that developers often undertake structural modifications to the codebase (e.g., reorganizing, optimizing, or simplifying code) during periods of performance evolution or change. This observation has two important implications: first, it underscores the role of refactorings as a potential driver or enabler of performance optimization; second, it highlights the likelihood that developers consider performance metrics when engaging in refactoring efforts. By identifying this connection, we establish a foundation for further exploration into whether specific refactoring activities correlate with, or can predict, meaningful performance outcomes. This insight could help inform better practices in code maintenance and performance engineering.

2. *Refactoring Type Distribution and Performance Change Correlations:* **Through our analysis of odds ratios in Table 4.3 we find that some refactorings such as Extract Method, Extract Variable, and Inline Method are more than three times more likely to impact performance compared to code without them (Odds ratio >3).** In contrast, method and parameter modifications showed notably lower associations, such as Add Method Annotation (0.24), Add Parameter (0.25), and Change Parameter Type (0.13) all had odds ratios below 0.5, suggesting these changes were less likely to influence performance changes. This pattern indicates that refactorings involving code restructuring had significantly stronger performance changes compared to interface or annotation modifications.

    - *Frequently Occurring Refactorings in Performance Context:* Extract Method and Extract Variable emerged as the most prevalent refactoring types in performance changes, with 61 and 49 occurrences respectively. When contextualized against their baseline frequencies in the complete dataset (4,223 and 3,356 occurrences), these refactoring types showed notably higher representation in performance changes (1.44% and 1.46% respectively). This disproportionate presence suggests these structural modifications play a significant role in performance changes.
    - *Moderate-Frequency Refactorings:* Change Variable Type demonstrated moderate frequency with 39 occurrences in performance changes. Despite its lower absolute count compared to Extract Method, its relative frequency (0.47% of total occurrences) indicates a consistent presence in refactorings leading to performance changes. This suggests that type-related refactorings, while less common overall, maintain relevance in performance optimization contexts.
    - *Low-Frequency but Significant Patterns:* Several refactoring types, including Inline

Table 4.3 Occurrences of Refactoring Types in Performance and Refactoring Datasets with Odds Ratio

| **Refactoring Type** | **Count in Performance Dataset** | **Count in Refactoring Dataset** | **Odds Ratio** |
|---|---|---|---|
| Extract Method | 61 | 4223 | 3.83 |
| Extract Variable | 49 | 3356 | 3.87 |
| Change Variable Type | 39 | 8332 | 1.24 |
| Inline Variable | 13 | 1380 | 2.50 |
| Add Method Annotation | 10 | 10898 | 0.24 |
| Inline Method | 9 | 669 | 3.57 |
| Add Variable Modifier | 7 | 1553 | 1.19 |
| Add Parameter | 6 | 6398 | 0.25 |
| Change Method Access Modifier | 6 | 6387 | 0.25 |
| Remove Method Annotation | 6 | 3743 | 0.42 |
| Change Parameter Type | 4 | 8115 | 0.13 |
| Remove Method Modifier | 2 | 1151 | 0.46 |

Method (9 occurrences) and Add Method Annotation (10 occurrences), showed lower absolute frequencies but maintained meaningful presence ratios when normalized against their baseline occurrences. The Inline Method's relative frequency of 1.35% indicates its particular significance in performance changes despite its lower absolute count.

*Refactorings frequently coincide with performance changes, with 80% of projects showing this overlap, including 20% of commits and 12.6% of methods. This indicates that refactorings often accompany performance change, suggesting their importance in code optimization practices. Notably, Extract Method and Extract Variable were the most common refactoring types likely to occur when there are performance changes, highlighting their role in performance tuning. The consistent involvement of method and variable manipulation refactorings, such as method granularity adjustments and variable scope refinements, emphasizes their contribution to both performance impact and code clarity during optimization efforts.*

**RQ2. How much does performance change when refactorings coincide with performance changes?**

**Motivation:**

Although it is useful to know which refactorings coincide with performance changes and how frequently these refactorings occur, it is equally important to understand the magnitude and direction of these performance changes. By quantifying how much performance changes (e.g., improvements or regressions) with each refactoring type, we can gain more actionable insights into their practical implications. Understanding the typical effect size distributions ranging from substantial performance gains or losses to neutral outcomes can inform developers about the potential risks and benefits of performing specific refactorings. Furthermore, it can guide future research and tool support, helping to detect and mitigate undesirable performance impacts.

**Approach:**

To analyze performance changes associated with refactorings, we collect execution distribution data before each method in our refactoring-related performance data point underwent refactoring and after. This data is obtained using our custom instrumentation tool, **JPerfEvo** (see 4.2).

For statistical analysis, we employed two complementary techniques: the *Mann-Whitney U-Test* [89] (see Appendix A) and *Cliff's Delta Effect Size* [90] (see Appendix A). The *Mann-Whitney U-Test* was used to assess the statistical significance of performance changes, determining whether observed differences between the before and after distributions (e.g., performance improvement or regression) were unlikely to have occurred by chance. This non-parametric test is well-suited for performance data, as it does not assume a normal distribution.

To complement this, we calculated *Cliff's Delta Effect Size* to quantify the magnitude of observed performance changes. This metric categorizes performance changes into four levels: negligible, small, medium, or large, providing a nuanced understanding of the practical significance of the observed differences.

Additionally, we computed the median percentage change in performance for each method as follows:

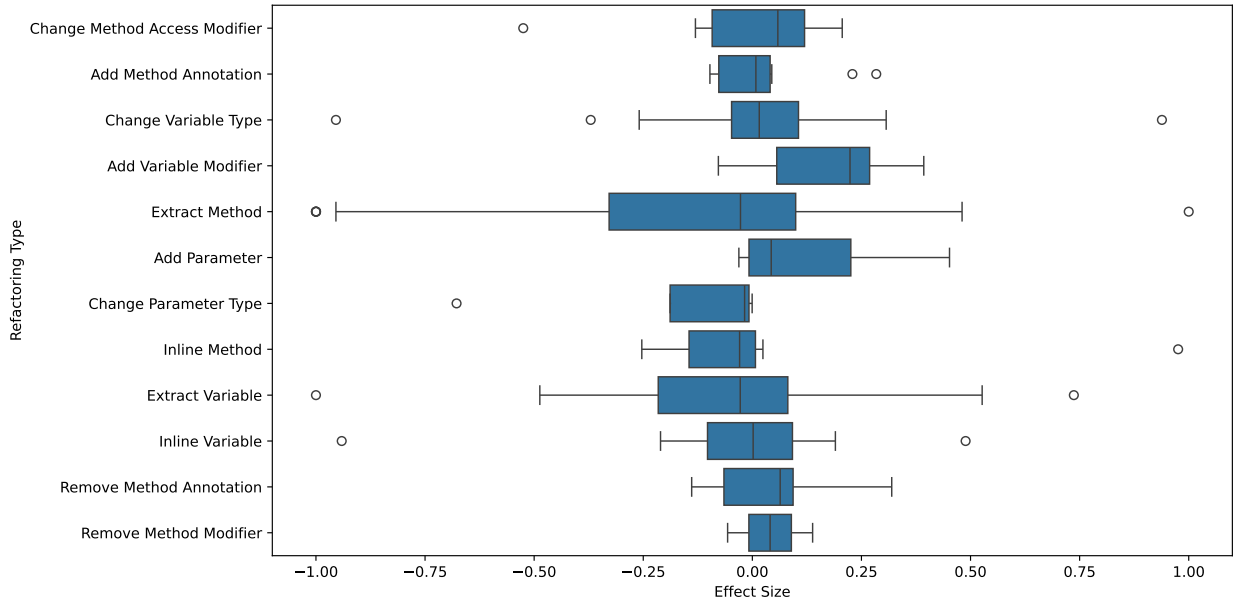


Figure 4.2 Performance changes effect size for each refactoring type.

$$\text{Median Percentage Change} = \frac{\text{Median}_{\text{after}} - \text{Median}_{\text{before}}}{\text{Median}_{\text{before}}} \times 100$$

This metric provides an intuitive representation of relative performance improvement or regression, offering a clearer interpretation of the method's evolution in the context of refactorings. Together, these robust statistical and descriptive analyses ensure a comprehensive and reliable detection of performance changes, supporting the broader goal of understanding the impact of refactorings on performance.

**Results:**

1. *Performance Change Effect Size in Each Refactoring Type*: **Figure 4.2 illustrates that the observed performance changes following different refactoring operations span a wide range, highlighting the inherent variability of their impacts. Notably, *Extract Method* (-0.954 to 0.481), and *Extract Variable* (-0.487 to 0.527) exhibit a broad spectrum of possible outcomes. These refactorings, traditionally associated with improving code clarity, can induce substantial regressions or yield meaningful improvements, depending on the context.** Such wide variability underscores the importance of careful evaluation and testing before implementing these changes at scale. Furthermore, operations

Table 4.4 Distribution of Effect Size Categories

| **Effect Size Category** | **Percentage (%)** |
|---|---|
| Negligible | 58.64 |
| Small | 23.04 |
| Large | 9.95 |
| Medium | 8.38 |

like *Inline Variable* (-0.210 to 0.190) and *Inline Method* (-0.253 to 0.024) also exhibit performance fluctuations, often leaning toward negative or neutral outcomes, thereby reinforcing the notion that certain refactoring types introduce heightened uncertainty in runtime behaviour.

In contrast, several refactoring types demonstrate relatively contained performance changes, suggesting more predictable runtime behaviour. *Add Method Annotation* operations exhibit a constrained range (-0.097 to 0.045), while *Change Parameter Type* modifications show minimal variation (-0.025 to 0.000). These constrained intervals imply that these operations, which are more closely tied to static code semantics, generally exert less extreme performance effects and may be safer choices for practitioners seeking stable runtime behaviour. Similarly, other refactoring types such as *Add Variable Modifier* (-0.078 to 0.393) and *Add Parameter* (-0.031 to 0.452) maintain bounded intervals, though their upper extremes suggest the potential for notable improvements under favourable conditions. While these refactoring types can occasionally boost performance, they remain comparatively predictable and manageable in terms of performance regression risk.

Finally, mid-range variability in operations such as *Change Variable Type* (-0.259 to 0.307) and *Remove Method Annotation* (-0.139 to 0.319) illustrates that even subtle syntactic or semantic modifications can shift performance in either direction. This observation suggests that routine and seemingly minor adjustments require careful validation to ensure that improvements in code maintainability or readability do not come at an unacceptable performance cost.

*While some refactorings such as Add Method Annotation and Change Parameter Type display limited variability and thus more predictable outcomes, others like Extract Method and Extract Variable exhibit broad, context-dependent ranges that can lead to both considerable performance gains and regressions. The variability observed in Change Variable Type and Remove Method Annotation, indicates that even minor code changes can influence performance outcomes. Recognizing these patterns is critical for practitioners seeking to balance code quality with the maintenance of stable, efficient runtime behaviour.*

2. *Distribution of Performance Change Effect Size* **The results shown in Table 4.4 indicate that the performance impact of over 40% of our sampled cases are non-negligible and could impact the state of practice.** While 23.04% of the observations result in small performance changes, they are still meaningful. Indeed, while these performance changes are not substantial, their frequency highlights that some refactoring types can induce subtle improvements or regressions that, over time, might accumulate into noticeable overall performance trends. Furthermore, the presence of large effects 9.95% and medium effects 8.38% demonstrates that certain refactorings can indeed produce pronounced performance changes. Therefore, while a majority of our sampled refactorings (58.64%) follow conventional refactoring wisdom and are correlated with negligible performance effects, these are far from representing all possible refactorings.

*Overall, the distribution of performance change effect sizes suggests that most refactorings result in negligible performance impact, with 58.64% of cases falling in this category. However, medium and large effects were observed 8.38% and 9.95% of the time, respectively. These insights underscore the importance of balancing code quality improvements with performance stability, as major performance changes associated with refactorings are not infrequent.*

**RQ3. What happens in refactorings when refactorings coincide with a performance change?**

**Motivation:**

Refactoring is essential for enhancing code readability, maintainability, and performance, yet its impact on performance can vary greatly across different refactoring types. While some refactorings yield minimal performance change, others lead to substantial performance changes underscoring the need to understand why such disparities arise. Factors like code complexity, underlying architecture, and the scale of modifications all influence whether a refactoring results in performance gains, regressions, or no change. By systematically examining these factors and the effect sizes of various refactoring operations, developers can make more informed decisions about when and how to refactor. This research question investigates what drives these performance changes and offers actionable insights into balancing performance improvements with long-term maintainability.

**Approach:**

1. *Data Classification:* We grouped our performance dataset based on *Cliff's Delta effect sizes* obtained from our approach in **RQ2**. While *Cliff's Delta* typically includes four categories (negligible, small, medium, and large), we focused on SMALL and LARGE effect sizes to clearly distinguish between minor and substantial performance changes. We excluded medium effect sizes to reduce ambiguity in our analysis, as the boundary between small-medium and medium-large changes could obscure clear patterns. Similarly, negligible effects were omitted as they indicated statistically insignificant performance changes. For each effect size category, we further classified changes based on their direction: performance improvement (positive change), regression (negative change), or unchanged. This focused classification approach provided a clear framework for identifying significant patterns and meaningful comparisons.

2. *Commit History Analysis:* For each data point in these groups, we manually examined their commit history in GitHub. This entailed reviewing commit messages, code snippets, associated descriptions, and any related GitHub issues. By scrutinizing these artifacts, we sought to uncover the underlying motivation for each refactoring operation, the specific refactoring types used, and contextual factors—such as code complexity, the system's architectural design, and the scope of modifications—that may have influenced performance. Where available, we also noted the developer's reasoning, as reflected in issue discussions or pull request comments, to gain insights into the decision-making process behind the refactoring effort.

3. *Performance Change Assessment:* This analysis enabled us to connect each refactoring

type with both its intended goals (e.g., simplifying code structure, improving readability, removing duplication) and its observed performance outcomes. We then interpreted these findings within the broader context of the software system, identifying whether performance gains or regressions could be linked to specific code-level changes, dependencies, or broader architectural constraints.

4. *Comparative Analysis and Pattern Identification:* With these individual data points examined, we conducted comparative analyses across entries that shared the same refactoring type and performance outcome but differed in terms of effect size (e.g., SMALL vs. LARGE). This aims to isolate and highlight the impacting factors ranging from the intricacy of the code being refactored to external system-level constraints that drove discrepancies in performance change. By systematically contrasting cases with similar goals yet divergent results, we aim to identify best practices and cautionary tales that illuminate when a particular refactoring is most likely to yield a substantial performance shift versus when its effects may be more minimal.

**Results**

Our analysis examines 35 data points to reveal the patterns, timing, and rationale behind refactoring efforts and their subsequent impact on performance change. We investigate cases where refactoring led to performance improvements, regressions or had no change, and the magnitude of these changes. Table 4.5 presents an overview of performance change types and effect sizes across different refactoring types. The chapter focuses on frequently occurring refactoring types and cases where multiple refactoring types were applied simultaneously. The full insight of our analysis can be found in Appendix B.

A. *Scenarios where performance improved after refactoring:*

See Appendix B for explanations and code snippets for other refactoring types.

1. The commit 69656a0 [12] in the *Elastic APM Java Agent* project involves refactoring the `startTransaction` method within the `ElasticApmTracer` class. Specifically, the refactoring includes extracting methods to enhance code readability and maintainability. These improvements notably delivered substantial performance gains.

[12]https://github.com/elastic/apm-agent-java/commit/69656a0a6185c659a0235f96c43bf7b30979a28e

Table 4.5 Overview of Performance Change Types and Effect Sizes for Refactoring Types

| **Refactorings** | **IL** | **IS** | **RL** | **RS** | **UCH** |
|---|---|---|---|---|---|
| Extract Method | 2 | 3 | 6 | 6 | 2 |
| Extract Variable | 2 | 3 | 1 | 6 | 5 |
| Change Variable Type | 1 | 3 | - | 1 | 1 |
| Inline Variable | 1 | - | 1 | 1 | 1 |
| Add Variable Modifier | - | 2 | - | - | 1 |
| Inline Method | 1 | - | - | 1 | - |
| Change Method Access Modifier | - | 1 | 1 | - | - |
| Add Method Annotation | - | 1 | - | - | 1 |
| Add Parameter | - | 1 | - | - | - |
| Change Variable Type + Add Variable Modifier | - | 1 | - | - | - |
| Extract Method + Remove Method Annotation | - | 1 | - | - | - |
| Extract Variable + Change Variable Type + Inline Variable | - | 1 | - | - | - |
| Extract Method + Change Parameter Type | - | - | 1 | - | - |
| Extract Method + Change Variable Type | - | - | 1 | - | - |
| Extract Method + Extract Variable | - | - | 1 | - | - |
| Inline Method + Extract Variable | - | - | - | 1 | - |
| Inline Variable + Extract Variable | - | - | - | 1 | - |

**Keys:** IL = Improvement_Large (Performance improvement with a large effect size), IS = Improvement_Small (Performance improvement with a small effect size), RL = Regression_Large (Performance regression with a large effect size), RS = Regression_Small (Performance regression with a small effect size), UCH = Unchanged (No performance change with a small effect size).

```
// Before Refactoring
public Transaction startTransaction() {
    // Original implementation with inlined logic
    if (isNoop()) {
        return noopTransaction();
    }
    Transaction transaction = new Transaction(/* parameters */);
    // Additional setup and initialization
    return transaction;
}

// After Refactoring
public Transaction startTransaction() {
    if (isNoop()) {
        return noopTransaction();
    }
    return createAndInitializeTransaction();
}

private Transaction createAndInitializeTransaction() {
    Transaction transaction = new Transaction(/* parameters */);
    // Additional setup and initialization
    return transaction;
}
```

The `startTransaction` method lies at a critical part within the codebase, operating on a hot path where performance is paramount. When refactored, even a slight reduction in overhead becomes magnified by the sheer number of invocations. This amplification effect means that seemingly small code changes can result in disproportionately large overall performance gains. Enhanced JVM inlining and optimization techniques play a crucial role in these improvements [91]. By breaking out the transaction creation into a dedicated method, the JVM can more effectively apply inlining optimizations. For hot methods, these optimizations can substantially reduce method call overhead, cut down on redundant instructions, and ultimately speed up execution. The architectural improvements extend to branch prediction as well. Separating the `noop` logic from the core transaction creation clarifies execution paths. This clarity allows the CPU's branch prediction mechanisms to work more efficiently, minimizing pipeline stalls and further

boosting performance. Complexity reduction has proven to be another key factor in achieving greater efficiency. Streamlining complex logic into smaller, self-contained methods not only makes the codebase easier to maintain but also gives the compiler and runtime clearer opportunities to optimize. When frequently executed methods are simpler, every call becomes more efficient. In summary, the large effect size is primarily driven by targeting a hot path with clearer, more modular code that the JVM and CPU can handle more efficiently. When code is executed at scale as APM agents typically are, the compounding effect of improved optimizations across thousands or millions of invocations underscores why this refactoring has had such a significant performance change.

2. The commit 8280252 [13] in the *Netty project* involves modifications to the `epollWait` method within the `Native` class. The changes include inlining methods to enhance performance, which led to a performance improvement.

```
// Before Refactoring
static int epollWait(FileDescriptor epollFd, EpollEventArray events,
↪  boolean immediatePoll) {
    // Original implementation with separate method calls
    int timeout = calculateTimeout(immediatePoll);
    return epollWaitNative(epollFd, events, timeout);
}

private static int calculateTimeout(boolean immediatePoll) {
    return immediatePoll ? 0 : -1;
}

private static native int epollWaitNative(FileDescriptor epollFd,
↪  EpollEventArray events, int timeout);

// After Refactoring
static int epollWait(FileDescriptor epollFd, EpollEventArray events,
↪  boolean immediatePoll) {
    // Inlined method calls for performance optimization
    int timeout = immediatePoll ? 0 : -1;
    return epollWaitNative(epollFd, events, timeout);
}
```

[13] https://github.com/netty/netty/commit/8280252d0e80e5e844215234732a7cda4862e260

```
private static native int epollWaitNative(FileDescriptor epollFd,
↪  EpollEventArray events, int timeout);
```

The seemingly minor refactoring of inlining the `calculateTimeout` method into `epollWait` creates substantial performance benefits due to several compounding factors. As a highly optimized I/O library, the `epollWait` method lies on a critical hot path, being called extremely frequently under heavy load conditions. In this context, even small overhead reductions multiply into significant performance gains across millions of invocations. The optimization works through multiple mechanisms. First, it eliminates method call overhead by removing the extra stack frame and potential CPU branch mispredictions that would occur with separate method calls [92]. Second, by presenting the JVM's JIT compiler with shorter, inlined methods, it enables more aggressive optimization opportunities, leading to improved throughput and lower latency during high-volume I/O operations. The consolidation of timeout logic into a single method also provides CPU-level benefits. The processor's branch prediction mechanisms can more effectively predict and pipeline the execution path (specifically for the `immediatePoll` condition), resulting in fewer pipeline stalls and better overall performance. This case demonstrates how targeted optimization of frequently executed code paths can yield outsized performance benefits, even when the changes appear minimal at first glance. The key is understanding where and how these optimizations will compound in production environments.

The most significant performance improvements consistently emerge when refactoring targets hot paths in the code - methods and operations that are frequently executed or exist in performance-critical sections. For example, in I/O operations like Netty's epollWait or APM agents' transaction monitoring, where even small optimizations compound dramatically due to high invocation rates.

The nature of the optimization itself plays a crucial role. Changes that reduce computational overhead, such as eliminating repeated method calls, improving memory access patterns, or enabling better JVM optimizations, tend to yield substantial benefits. This is particularly evident in cases like replacing boxed types with primitives, utilizing unsafe buffers for faster memory access, or strategically inlining methods to reduce call overhead and improve JIT compilation.

Architectural context heavily influences the magnitude of performance gains. Refactorings that simplify frequently traversed code paths, improve CPU branch prediction or reduce

garbage collection pressure often show larger improvements. However, when changes target infrequently executed code or focus primarily on maintainability (such as extracting methods for clarity or adding final modifiers), the performance change tends to be modest regardless of the optimization's theoretical benefits.

> *The compounding effect of optimizations in high-throughput scenarios emerges as a critical factor. When refactored code executes thousands or millions of times, even minor improvements in efficiency can accumulate into significant performance gains. This explains why seemingly small changes like extracting a length variable in a tight loop or optimizing stack trace processing can have outsized effects when applied to frequently executed paths.*

B. *Scenarios where performance regressed after refactoring:*

See Appendix B for explanations and code snippets for other refactoring types.

1. The commit 940e717 [14] in the *Elastic APM Java Agent* project involves refactoring the `fillStackTrace` method within the `CurrentThreadStackTraceFactory` class. Specifically, the refactoring includes extracting a new method to enhance code readability and maintainability. However, this change resulted in a significant performance regression.

```
// Before Refactoring
public void fillStackTrace(List<Stacktrace> stacktrace) {
    // Original implementation with inlined logic
    // ...
    for (int i = 0; i < stackTraceElements.length; i++) {
        // Complex logic directly within the loop
        // ...
    }
    // ...
}

// After Refactoring
public void fillStackTrace(List<Stacktrace> stacktrace) {
    // Refactored implementation with extracted method
    // ...
```

[14] https://github.com/elastic/apm-agent-java/commit/940e717ad9662d5009f6e6b6bd14f66c5bd896f0

```
    for (int i = 0; i < stackTraceElements.length; i++) {
        processStackTraceElement(stackTraceElements[i], stacktrace);
    }
    // ...
}

private void processStackTraceElement(StackTraceElement element,
↪  List<Stacktrace> stacktrace) {
    // Logic extracted from the original loop
    // ...
}
```

The seemingly beneficial refactoring of extracting `processStackTraceElement` from the `fillStackTrace` loop led to a significant performance regression due to its placement in a performance-critical path. While the extraction improved code readability and maintainability, it introduced considerable overhead in monitoring systems where stack trace processing is a frequently executed operation. The performance change stems from three key factors. First, adding a separate method call for each stack trace element created substantial overhead within a high-volume loop, particularly impactful in time-sensitive APM agent operations. Second, the complexity of the extracted logic likely exceeded the JVM's inlining thresholds, preventing the runtime optimizations that typically occur with simpler, inlined code. Finally, since stack trace capture and processing is a hot path in monitoring agents, these small overhead increases compound rapidly under production load. This case illustrates an important principle in performance-sensitive systems: method extractions that would be harmless in most contexts can become significant bottlenecks when they affect frequently executed code paths, highlighting the careful balance needed between code maintainability and runtime performance.

2. In commit 70ea034[15] in the *SimpleFlatMapper* project, the `finish` method in the `CharConsumer` class underwent refactoring, specifically involving method inlining.

```
// Before Refactoring
public final void finish(CellConsumer cellConsumer) {
    endOfRow(cellConsumer);
```

[15]https://github.com/arnaudroger/SimpleFlatMapper/commit/70ea034333aa47e5e54753f764ae4d493f13e03d

```
}

// After Refactoring
public final void finish(CellConsumer cellConsumer) {
    cellConsumer.endOfRow();
}
```

---

The attempt to optimize performance through manual inlining of the `endOfRow(cellConsumer)` method actually resulted in a small performance regression, contrary to typical inlining benefits. This unexpected outcome occurred because the JVM's sophisticated runtime optimization already efficiently handled such small, simple methods through automatic inlining. The original method's limited scope and straightforward nature meant there was not only little room for performance improvement, but the manual intervention disrupted the JVM's existing optimization patterns. The refactoring introduced two disadvantages: it reduced code readability by removing a clearly named, focused method, and it created a minor performance regression by interfering with the JVM's natural optimization process. This illustrates an important principle in optimization: not all inlining efforts produce improvements, and some can result in actual performance degradation while making the code less maintainable. The case demonstrates how existing JVM optimizations often handle simple cases more effectively than manual intervention, making such changes both unnecessary and potentially harmful to performance.

The most significant impact of performance regression occurs when refactoring introduces additional method calls in frequently executed code paths, particularly in performance-critical components like monitoring agents, statistical computations, or high-throughput operations. This overhead becomes especially pronounced when the extracted methods are complex enough to prevent effective JVM inlining optimizations.

Data structure changes can also substantially impact performance, as seen in cases where switching from specialized structures (like Lists) to more generic ones (like Collections) sacrifices efficient operations such as random access. Similarly, removing cached variables in favour of repeated method calls can degrade performance, especially when those methods involve non-trivial work or synchronization.

The impact of these refactorings is often magnified in hot code paths where operations occur frequently. Even seemingly minor changes, such as introducing additional local variables or

altering JVM caching patterns, can accumulate to create noticeable performance regressions in high-throughput scenarios. This is particularly evident in systems handling intensive data processing or requiring ultra-low latency.

> *When refactoring focuses purely on structural changes that do not alter the underlying computation patterns or data access methods such as extracting methods for better organization without adding complexity - the performance change tends to be minimal or non-existent. This suggests that performance regressions are most likely when refactoring changes not just the code's structure but also its execution patterns or resource utilization characteristics.*

C. *Scenarios where performance unchanged after refactoring:*

See Appendix B for explanations and code snippets for other refactoring types.

1. In commit 259ebf5 [16] in the project , the `waitForMetadata` method in the `DslJsonSerializer` class underwent a refactoring that involved inlining a variable to streamline the code.

```
// Before Refactoring
private void waitForMetadata() {
    final long metadataRefreshInterval =
    ↪  coreConfiguration.metadataRefreshInterval.getMillis();
    // ... existing code ...
}

// After Refactoring
private void waitForMetadata() {
    // Directly using the method call without assigning it to a variable
    // ... existing code ...
}
```

   The refactoring of `waitForMetadata` focused on simplifying the code by directly using `coreConfiguration.metadataRefreshInterval.getMillis()` instead of storing it in an intermediate variable `metadataRefreshInterval`. While this change made the code more concise, it had no measurable impact on performance for several key reasons.

[16] https://github.com/elastic/apm-agent-java/commit/259ebf52dac8ea5136e54b6aecb246b3ffe8fe2c

Modern Java compilers are highly efficient at optimizing simple code patterns like variable assignments and method calls [93, 94]. In this case, since `getMillis()` is likely a straightforward accessor method, the compiler could apply the same optimizations regardless of whether the value was stored in an intermediate variable or used directly. The removal of a single variable assignment was too minimal to affect execution flow or resource usage in any meaningful way. This case demonstrates how certain code cleanups can successfully improve readability without compromising performance. The unchanged effect size confirms that developers can confidently make such simplifying refactors in similar scenarios where the changes primarily affect basic code structure rather than execution patterns.

2. In commit a53d152 [17] in the *JDBI* project, the `findGenericParameter` method in the `GenericTypes` class was updated to include the `@Nullable` annotation on its return type.

```
// Before Refactoring
public static Optional<Type> findGenericParameter(Type type, Class<?>
↪  parameterizedSupertype, int n) {
    // Method implementation
}

// After Refactoring
@Nullable
public static Optional<Type> findGenericParameter(Type type, Class<?>
↪  parameterizedSupertype, int n) {
    // Method implementation
}
```

The addition of the `@Nullable` annotation to the `findGenericParameter` method had no impact on runtime performance, as annotations are purely compile-time constructs. However, the change brought several developmental benefits: it explicitly communicates the possibility of null returns to developers, enables better static analysis for catching potential null dereference issues, and serves as clear documentation of the method's contract. This case illustrates how certain code improvements can enhance maintainability and safety without affecting runtime performance. The `@Nullable` annotation

[17] https://github.com/jdbi/jdbi/commit/a53d152527e7faa4e91553a0625235024b2da862

represents a best practice in Java development that helps prevent bugs through better communication of method behaviour [95] while maintaining the same execution efficiency.

Code changes focused purely on compile-time behaviours, such as adding annotations or improving documentation, tend to have no runtime impact since they only affect development-time tools and code analysis. These modifications enhance code clarity and maintainability without altering the execution path. Memory-focused refactorings that modify allocation strategies or resource management patterns can maintain consistent performance while improving efficiency. For instance, switching from fixed pre-allocation to dynamic sizing changes how resources are utilized without affecting the computational speed of operations performed on those resources. Structural code changes that reorganize logic without modifying the underlying algorithms also preserve performance characteristics. Examples include inlining variables or consolidating related functionality, where the same operations occur but in a more organized or maintainable format. These changes primarily benefit code readability and maintenance without introducing computational overhead. Functional extensions that maintain the same processing approach while supporting new use cases can also achieve unchanged performance. When such changes preserve the original algorithmic complexity and only add handling for new formats or edge cases, they expand functionality without degrading performance. This is particularly true when the core processing logic remains consistent and any new handling follows the same computational patterns as the original code.

*Compile-time changes like annotations and documentation improve clarity without affecting runtime. Memory optimizations, such as dynamic allocation, enhance resource usage without changing performance. Structural refactors reorganize code for maintainability without impacting efficiency. Functional extensions that preserve algorithmic patterns can add new features without degrading performance. These changes focus on improving readability and resource management while maintaining computational efficiency.*

## 4.4 Conclusion

This chapter explores the relationship between refactorings and software performance, shedding light on how various refactoring operations influence runtime behaviour across 15 open-source Java projects. Our study employs a systematic design, combining a comprehensive dataset of refactorings and performance benchmarks with automated tools for code change detection and performance measurement. By using JPerfEvo[1] a custom benchmark pipeline tool, in combination with RefactoringMiner we can mine our studied projects for refactor-

ings to build datasets, identify code changes, filter performance-related commits, and measure execution time variations under controlled conditions. We thus analyse the extent and refactorings coinciding with performance changes. Our findings reveal that method extraction and variable inlining are the most common refactorings associated with performance changes in our dataset, with both positive and negative performance implications depending on code context. We also analyse the magnitude of performance changes during refactorings, where our analysis shows that while certain refactorings lead to significant performance gains, others introduce regressions, particularly when complex code paths are involved. Finally, we investigate contextual factors influencing performance changes, and our results emphasize that refactoring outcomes depend heavily on the project's code structure, frequency of method invocations, and the extent of changes applied.

# CHAPTER 5 THREATS TO VALIDITY

Threats to validity refer to potential limitations in our study that may impact the reliability and generalizability of our findings. We categorize these threats into **construct validity**, **internal validity**, and **external validity** concerns.

**Construct Validity:** Construct validity threats arise when the measurements used in the study do not fully capture the intended concepts. In our study, we rely on RefactoringMiner, a widely used refactoring detection tool, to extract refactoring operations. While RefactoringMiner achieves high precision (99.8%) and recall (98.1–98.2%), there remains a possibility of false positives and missed refactorings. To mitigate this, we manually verify a subset of the detected refactorings.

Additionally, we categorize performance impacts using standard Cliff's Delta effect size thresholds, which are widely validated but may still lead to misclassification of performance changes (e.g., small, medium, or large) due to dataset variability. Furthermore, our study primarily evaluates execution time as the performance metric, potentially overlooking other important factors such as CPU utilization and memory consumption, which could provide a more comprehensive view of performance fluctuations. Future work should incorporate multiple performance metrics to enhance measurement accuracy.

Another construct validity concern arises from potential bias in performance bug collection, as researcher judgment may influence issue selection and categorization. To minimize this risk, we conducted post-classification discussions between two researchers to resolve any discrepancies in labelling.

**Internal Validity:** Internal validity threats refer to confounding factors that could provide alternative explanations for observed results. One key concern is the classification of performance-related labels in our study. While the two authors independently identified and discussed these labels, it is possible that our dataset did not capture all potential categories. To mitigate this, we ensured a thorough labelling process for all extracted performance issues, reducing the likelihood of missing key classifications.

Another potential issue is the accuracy of performance measurements. We used a lightweight instrumentation tool that underwent multiple validation tests to ensure precision and minimal overhead. However, noise in the dataset remains a possibility, and execution measurements could be influenced by system fluctuations. Moreover, benchmarking was conducted on Google Cloud Platform's virtual machines, where hypervisor resource-sharing mechanisms

could introduce performance variability despite resource allocation guarantees. Future studies should consider using dedicated hardware or quantifying the impact of virtualization on performance results.

Furthermore, our study includes manual analyses, which inherently introduce a degree of subjectivity. To mitigate this, we followed standard research practices by conducting multiple rounds of analysis with two independent reviewers. All results, methodologies, and classification criteria are documented in our replication package to enhance transparency and reproducibility.

Additionally, a significant number of commits were excluded from the benchmarking process due to build failures or insufficient microbenchmark coverage for modified methods. This exclusion may have led to some performance-impacting changes being omitted from the final analysis.

**External Validity:** External validity threats concern the generalizability of our findings beyond the studied dataset. We selected 15 Java open-source projects of varying sizes, complexities, and domains to ensure diversity. However, there is no guarantee that our findings extend to proprietary software or projects with different maintenance and development practices. Additionally, the performance characteristics observed in Java may not be representative of other programming languages, particularly due to differences in JVM optimizations and language-specific performance behaviors. Future research should investigate whether similar trends hold in other ecosystems.

We also ensured that the selected projects were actively maintained and had well-documented performance issues available on GitHub. However, this means that performance-critical optimizations typically found in enterprise or proprietary software may not have been fully captured. Lastly, newer Java versions may introduce performance optimizations that were not accounted for in our study, affecting the relevance of some findings over time.

# CHAPTER 6 CONCLUSION

Software performance is a crucial aspect of modern software systems, directly impacting user experience, resource efficiency, and overall system reliability. Despite the well-established role of refactoring in improving code maintainability, readability, and extensibility, its relationship with software performance remains under-explored. This thesis bridges this gap by presenting two complementary studies that investigate the role of refactoring in addressing performance-related issues and its impact on execution time. This chapter presents a comprehensive conclusion to the thesis, outlining its key contributions and findings, discussing limitations, and suggesting avenues for future research.

## 6.1 Summary of Works

In Chapter 3, we examined 255 open-source Java projects, analyzing performance-related issues, commits, and pull requests using GitHub's API and RefactoringMiner. Our findings reveal that performance issues are primarily addressed through class restructuring and method modifications, with "Change Variable Type" emerging as the most frequently applied refactoring type. Statistical analysis indicates that performance-related commits are 1.77 times more likely to contain refactoring operations than non-performance commits, highlighting a strong correlation between performance concerns and refactoring activities. Furthermore, a manual classification of 300 performance-related refactorings identified 11 distinct motivations for refactoring in performance-critical contexts and three primary refactoring application contexts.

In Chapter 4, we focused on the execution-time impact of refactoring in 15 open-source Java projects, utilizing an automated performance benchmarking pipeline. By leveraging tools such as JPerfEvo, the Java Microbenchmark Harness (JMH), and a JVM instrumentation agent, we systematically measured the execution time variations introduced by different refactoring operations. Our results indicate that method extraction and variable inlining are the refactorings most frequently associated with execution-time variations, with certain refactoring types contributing to significant performance gains, while others occasionally introduce regressions. Contextual factors such as method invocation frequency, code complexity, and test coverage emerged as key determinants of performance outcomes.

## 6.2 Contributions and Implications

This research makes several key contributions to the understanding of refactoring's role in performance optimization:

1. **Empirical Evidence on Performance-Related Refactoring:** We provide a large-scale analysis of how refactoring is employed to resolve performance issues in open-source Java projects, identifying common refactoring types and their prevalence in performance-critical commits.
2. **Execution Time Impact Assessment:** By measuring performance changes across multiple refactored versions of software, we offer insights into which refactoring types are most likely to impact execution time, enabling developers to make informed refactoring decisions in performance-sensitive applications.
3. **Refactoring Motivation and Context Classification:** We introduce a classification scheme for developer motivations behind performance refactorings, shedding light on when and why developers refactor code to improve performance.
4. **Benchmarking Methodology for Performance Refactoring:** We propose a replicable methodology for measuring refactoring-induced performance changes using JMH-based benchmarking and JVM instrumentation, setting a foundation for future research in automated performance evaluation.

## 6.3 Limitations and Future Work

While this research provides valuable insights, several limitations pave the way for future exploration:

1. **Language and Ecosystem Scope:** Our study focuses on Java open-source projects; extending the analysis to other languages (e.g., Python, C++, Rust) could provide a broader perspective on refactoring's impact across diverse ecosystems.
2. **Granularity of Performance Measurement:** Although we examined execution time, future studies could explore additional performance metrics such as memory usage, CPU utilization, and I/O latency to develop a more holistic understanding of refactoring's performance impact.

3. **Automated Performance-Aware Refactoring Tools:** Given our findings, an interesting direction for future work is the development of intelligent refactoring tools that provide performance-aware recommendations, helping developers optimize software without introducing unintended performance regressions.

4. **Industry Adoption and Case Studies:** While our research is grounded in open-source repositories, future studies could investigate how performance refactoring is applied in industrial settings, incorporating developer feedback and real-world use cases.

# REFERENCES


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

[2] A. Nistor, T. Jiang, and L. Tan, "Discovering, reporting, and fixing performance bugs," 05 2013, pp. 237–246.

[3] Y. Zhao, L. Xiao, X. Wang, L. Sun, B. Chen, Y. Liu, and A. B. Bondi, "How are performance issues caused and resolved?-an empirical study from a design perspective," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 181–192.

[4] V. Cortellessa, D. D. Pompeo, V. Stoico, and M. Tucci, "On the impact of performance antipatterns in multi-objective software model refactoring optimization," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2021, pp. 224–233.

[5] D. Arcelli, V. Cortellessa, and C. Trubiani, "Antipattern-based model refactoring for software performance improvement," in *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, ser. QoSA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 33–42. [Online]. Available: https://doi.org/10.1145/2304696.2304704

[6] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[7] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[8] M. Fowler and K. Beck, "Refactoring: Improving the design of existing code," in *11th European Conference. Jyväskylä, Finland*, 1997.

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

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-oriented reengineering patterns.* Elsevier, 2002.

[11] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. Van Gorp, "Refactoring: Current research and future trends," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 3, pp. 483–499, 2003.

[12] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 858–870. [Online]. Available: https://doi.org/10.1145/2950290.2950305

[13] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[14] W. F. Opdyke, *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign, 1992.

[15] E. Zabardast, J. Gonzalez-Huerta, and D. Šmite, "Refactoring, bug fixing, and new development effect on technical debt: An industrial case study," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 376–384.

[16] A. Vasileva and D. Schmedding, "How to improve code quality by measurement and refactoring," in *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, 2016, pp. 131–136.

[17] G. Szőke, C. Nagy, R. Ferenc, and T. Gyimóthy, "A case study of refactoring large-scale industrial systems to efficiently improve source code quality," in *Computational Science and Its Applications–ICCSA 2014: 14th International Conference, Guimarães, Portugal, June 30–July 3, 2014, Proceedings, Part V 14*. Springer, 2014, pp. 524–540.

[18] M. Mohan, D. Greer, and P. McMullan, "Technical debt reduction using search based automated refactoring," *Journal of Systems and Software*, vol. 120, pp. 183–194, 2016.

[19] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

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

[21] M. Agnihotri and A. Chug, "Understanding the effect of batch refactoring on software quality," *International Journal of System Assurance Engineering and Management*, pp. 1–9, 2024.

[22] A. Imran, T. Kosar, J. Zola, and M. F. Bulut, "Predicting the impact of batch refactoring code smells on application resource consumption," *arXiv preprint arXiv:2306.15763*, 2023.

[23] W. Chen, P. Chang, T. Conte, and W. Hwu, "The effect of code expanding optimizations on instruction cache design," *IEEE Transactions on Computers*, vol. 42, no. 9, pp. 1045–1057, 1993.

[24] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 352–361.

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

[26] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[27] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[28] H. Xie *et al.*, "Research and case analysis of apriori algorithm based on mining frequent item-sets," *Open Journal of Social Sciences*, vol. 9, no. 04, p. 458, 2021.

[29] F. Coelho, N. Tsantalis, T. Massoni, and E. L. G. Alves, "An empirical study on refactoring-inducing pull requests," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3475716.3475785

[30] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes? a multi-project study," in *Proceedings of the XXXI Brazilian Symposium on Software Engineering*, ser. SBES '17. New York,

NY, USA: Association for Computing Machinery, 2017, p. 74–83. [Online]. Available: https://doi.org/10.1145/3131151.3131171

[31] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, "Why developers refactor source code: A mining-based study," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, sep 2020. [Online]. Available: https://doi.org/10.1145/3408302

[32] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[33] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. Ferreira, and D. Dig, "30 years of software refactoring research: A systematic literature review," *arXiv preprint arXiv:2007.02194*, 2020.

[34] S. M. Akhtar, M. Nazir, A. Ali, A. S. Khan, M. Atif, and M. Naseer, "A systematic literature review on software-refactoring techniques, challenges, and practices," *VFAST Transactions on Software Engineering*, vol. 10, no. 4, pp. 93–103, 2022.

[35] M. Alotaibi and M. W. Mkaouer, "Advances and challenges in software refactoring: A tertiary systematic literature," *Academia*, 2018.

[36] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," *Journal of Software Engineering Research and Development*, vol. 6, pp. 1–52, 2018.

[37] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: a systematic literature review," *Software Quality Journal*, vol. 28, no. 2, pp. 459–502, 2020.

[38] Y. Liu, "Software refactoring and rewriting: from the perspective of code transformations," *arXiv preprint arXiv:2308.06615*, 2023.

[39] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "How do i refactor this? an empirical study on refactoring trends and topics in stack overflow," *Empirical Software Engineering*, vol. 27, no. 1, p. 11, 2022.

[40] A. Almogahed, M. Omar, and N. H. Zakaria, "Recent studies on the effects of refactoring in software quality: Challenges and open issues," in *2022 2nd International Conference on Emerging Smart Technologies and Applications (eSmarTA)*. IEEE, 2022, pp. 1–7.

[41] S. Motogna, L.-M. Berciu, and V.-A. Moldovan, "Artificial intelligence methods in software refactoring: A systematic literature review," in *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2024, pp. 309–316.

[42] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, "A survey of deep learning based software refactoring," *arXiv preprint arXiv:2404.19226*, 2024.

[43] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One thousand and one stories: a large-scale survey of software refactoring," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1303–1313.

[44] E. A. AlOmar, T. Wang, V. Raut, M. W. Mkaouer, C. Newman, and A. Ouni, "Refactoring for reuse: an empirical study," *Innovations in Systems and Software Engineering*, vol. 18, no. 1, pp. 105–135, 2022.

[45] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: Performance bug detection in the wild," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 155–170. [Online]. Available: https://doi.org/10.1145/2048066.2048081

[46] S. Li, "Instantaneous performance bug detection in ide," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 692–694.

[47] R. Krasniqi, "Extractive summarization of related bug-fixing comments in support of bug repair," in *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, 2021, pp. 31–32.

[48] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Fixing performance bugs: An empirical study of open-source gpgpu programs," in *2012 41st International Conference on Parallel Processing*, 2012, pp. 329–339.

[49] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 556–567. [Online]. Available: https://doi.org/10.1145/3368089.3409695

[50] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on firefox," in *Proceedings of the 8th working conference on mining software repositories*, 2011, pp. 93–102.

[51] J. Imseis, C. Nachuma, S. Arifuzzaman, and Z. Bhuiyan, *On the Assessment of Security and Performance Bugs in Chromium Open-Source Project*, 11 2019, pp. 145–157.

[52] A. Rajbhandari, M. F. Zibran, and F. Z. Eishita, "Security versus performance bugs: How bugs are handled in the chromium project," in *2022 IEEE/ACIS 20th International Conference on Software Engineering Research, Management and Applications (SERA)*, 2022, pp. 70–76.

[53] L. Kumar, S. Tummalapalli, and L. Murthy, "An empirical framework to investigate the impact of bug fixing on internal quality attributes," *Arabian Journal for Science and Engineering*, vol. 46, 11 2020.

[54] H. Anwar, D. Pfahl, and S. N. Srirama, "Evaluating the impact of code smell refactoring on the energy consumption of android applications," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2019, pp. 82–86.

[55] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption," in *ICSOFT 2021 - 16th International Conference on Software Technologies*, Virtual, France, Jul. 2021. [Online]. Available: https://hal.science/hal-03202437

[56] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2652524.2652538

[57] E. Iannone, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Refactoring android-specific energy smells: A plugin for android studio," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 451–455. [Online]. Available: https://doi.org/10.1145/3387904.3389298

[58] B. A. Muse, F. Khomh, and G. Antoniol, "Do developers refactor data access code? an empirical study," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 25–35.

[59] S. H. Kannangara and W. M. J. I. Wijayanayake, "Impact of refactoring on external code quality improvement: An empirical evaluation," in *2013 International Conference on Advances in ICT for Emerging Regions (ICTer)*, 2013, pp. 60–67.

[60] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 284–294.

[61] A. Imran and T. Kosar, "The impact of auto-refactoring code smells on the resource utilization of cloud software," *arXiv preprint arXiv:2008.06214*, 2020.

[62] M. M. U. Alam, J. Gottschlich, and A. Muzahid, "Autoperf: A generalized zero-positive learning system to detect software performance anomalies," *arXiv preprint arXiv:1709.07536*, 2017.

[63] M. Peiris and J. H. Hill, "Towards detecting software performance anti-patterns using classification techniques," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–4, 2014.

[64] T. Parsons, "A framework for detecting, assessing and visualizing performance antipatterns in component based systems," in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004, pp. 316–317.

[65] Y. Zhao, L. Xiao, C. Wei, R. Kazman, and Y. Yang, "A systematic mapping study on architectural approaches to software performance analysis," *arXiv preprint arXiv:2410.17372*, 2024.

[66] S. S. N. Gaonkar, A. Pai, and L. M. Colaco, "Performance testing and enhancement of java web applications," *International Journal of System & Software Engineering*, vol. 7, no. 1, pp. 28–37, 2019.

[67] B. Li, P. Su, M. Chabbi, S. Jiao, and X. Liu, "Djxperf: Identifying memory inefficiencies via object-centric profiling for java," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, 2023, pp. 81–94.

[68] V. Cortellessa, A. Di Marco, and C. Trubiani, "An approach for modeling and detecting software performance antipatterns based on first-order logics," *Software & Systems Modeling*, vol. 13, pp. 391–432, 2014.

[69] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software performance testing based on workload characterization," in *Proceedings of the 3rd International Workshop on Software and Performance*, 2002, pp. 17–24.

[70] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on software Engineering*, vol. 18, no. 5, p. 423, 1992.

[71] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in java-based open source projects," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 373–384.

[72] L. Traini, F. Di Menna, and V. Cortellessa, "Ai-driven java performance testing: Balancing result quality with testing time," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 443–454.

[73] L. Traini, V. Cortellessa, D. Di Pompeo, and M. Tucci, "Towards effective assessment of steady state performance in java software: Are we there yet?" *Empirical Software Engineering*, vol. 28, no. 1, p. 13, 2023.

[74] M. Niranjanamurthy, A. Saha, D. Chahar *et al.*, "Comparative study on performance testing with jmeter," *Int. J. Adv. Res. Comput. Commun. Eng*, vol. 5, no. 2, pp. 70–76, 2016.

[75] P. Stefan, V. Horky, L. Bulej, and P. Tuma, "Unit testing performance in java projects: Are we there yet?" in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 401–412.

[76] W. Ma, L. Chen, Y. Zhou, and B. Xu, "What are the dominant projects in the github python ecosystem?" in *2016 Third International Conference on Trustworthy Systems and their Applications (TSA)*, 2016, pp. 87–95.

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

[78] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Tandem: A taxonomy and a dataset of real-world performance bugs," *IEEE Access*, vol. 8, pp. 107 214–107 228, 2020.

[79] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT*

*International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 155–165. [Online]. Available: https://doi.org/10.1145/2635868.2635922

[80] M. L. McHugh, "The chi-square test of independence," *Biochemia medica*, vol. 23, no. 2, pp. 143–149, 2013.

[81] J. M. Bland and D. G. Altman, "The odds ratio," *Bmj*, vol. 320, no. 7247, p. 1468, 2000.

[82] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[83] E. A. AlOmar, J. Liu, K. Addo, M. W. Mkaouer, C. Newman, A. Ouni, and Z. Yu, "On the documentation of refactoring types," *Automated Software Engineering*, vol. 29, pp. 1–40, 2022.

[84] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "Scottknott: a package for performing the scott-knott clustering algorithm in r," *TEMA (São Carlos)*, vol. 15, pp. 3–17, 2014.

[85] C. Laaber, S. Würsten, H. C. Gall, and P. Leitner, "Dynamically reconfiguring software microbenchmarks: Reducing execution time without sacrificing result quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 989–1001.

[86] N. Aaraj, A. Raghunathan, S. Ravi, and N. K. Jha, "Energy and execution time analysis of a software-based trusted platform module," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

[87] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International conference on software maintenance*. IEEE, 2013, pp. 516–519.

[88] J. M. Bland and D. G. Altman, "The odds ratio," *Bmj*, vol. 320, no. 7247, p. 1468, 2000.

[89] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[90] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[91] J. Shirazi. (2023) Java performance tuning tips. Accessed: 2025-01-10. [Online]. Available: https://www.javaperformancetuning.com/tips/rawtips.shtml

[92] D. Csákvári. (2015) Jvm jit optimization techniques. Accessed: 2025-01-11. [Online]. Available: https://advancedweb.hu/jvm-jit-optimization-techniques/

[93] M. Aibin. (2024) Method inlining in the jvm. Accessed: 2025-01-19. [Online]. Available: https://www.baeldung.com/jvm-method-inlining

[94] A. C. Codex. (2024) Java compiler optimization: Advanced features. Accessed: 2025-01-19. [Online]. Available: https://reintech.io/blog/java-compiler-optimization-advanced-features

[95] Spring-Framework-Docs. (2023) Null safety and the `@Nullable` annotation. Accessed: 2025-01-19. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/null-safety.html

[96] C. O. Fritz, P. E. Morris, and J. J. Richler, "Effect size estimates: current use, calculations, and interpretation." *Journal of Experimental Psychology: General*, vol. 141, pp. 2–18, 2012.

[97] P. Krill. (2023) Make java fast: Optimize your code for performance. Accessed: 2025-01-10. [Online]. Available: https://www.infoworld.com/article/2174629/make-java-fast-optimize.html

[98] Y. Ishikawa, H. Komatsu, and T. Nakatani, "Improving code caching performance for java applications," in *Proceedings of the 2008 ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2008, pp. 177–186.

# APPENDIX A DETAILED EXPLANATIONS OF EQUATIONS REQUIRED FOR REPLICATION

### A.0.1 Detailed Explanation of Odds Ratio Variables

The variables in the Odds Ratio equation (Equation 3.3) are defined as follows:

- $a$: Count of performance-related commits where refactoring is present.
- $b$: Count of performance-related commits where refactoring is absent.
- $c$: Count of non-performance-related commits where refactoring is present.
- $d$: Count of non-performance-related commits where refactoring is absent.

**A.0.2 Detailed Explanation of Snott-Knott Test**// The Scott-Knott Test addresses the limitations of traditional post-hoc tests by creating statistically distinct and non-overlapping groups, which is particularly valuable when dealing with numerous refactoring types. The Scott-Knott (SK) procedure is a hierarchical clustering algorithm that works with balanced designs, where we have a set of independent sample treatment means, each with the same number of replications, all normal variates [84]. The algorithm operates as follows: (1) It starts by treating all refactoring types as a single group. (2) For each possible division of the group into two subgroups, it calculates the between-group sum of squares ($B$). (3) It identifies the division that maximizes $B$, denoted as $B_o$. (4) It applies a likelihood ratio test to determine if the division is statistically significant. (5) If the division is significant, the algorithm is recursively applied to each of the resulting subgroups. (6) The process continues until no significant divisions can be made or until each group contains only one observed mean. The $p$-value of the likelihood ratio test can be interpreted as a distance between the two selected groups, with the chosen significance level ($\alpha$) serving as the cutoff. If the $p$-value is smaller than $\alpha$, the groups are considered heterogeneous and should be separated; otherwise, they form a homogeneous group. The Scott-Knott Test helped us achieve several important objectives: (1) Identification of Distinct Groups: It allowed us to identify groups of refactoring types that are statistically distinct from each other in terms of their frequency or importance across projects. (2) Reduction of Subjective Interpretation: Providing non-overlapping groups, reduced the need for subjective interpretation of pairwise comparisons. (3) Hierarchical Structure: The resulting grouping provided a hierarchical structure of refactoring types, offering insights into their relative importance or frequency. (4)Simplification

of Results: It simplified the presentation and interpretation of results, especially valuable given our large number of refactoring types.

Refactoring types within the same group are not significantly different from each other but are significantly different from those in other groups. This grouping provides a clear ranking of refactoring types based on their prevalence or importance in addressing performance issues across projects.

The variables in the likelihood ratio test equation (Equation 3.6) are defined as follows:

- $\lambda$: Test statistic for the likelihood ratio test, asymptotically $\chi^2$ distributed.
- $\pi$: A constant in the equation (specific value not provided in the original text).
- $B_o$: A parameter in the equation (specific meaning not provided in the original text).
- $\sigma_o^2$: The maximum likelihood estimator of $\sigma_r^2$.
- $\nu_o$: Degrees of freedom for the $\chi^2$ distribution, equal to $k(\pi - 2)$.
- $k$: A parameter in the degrees of freedom calculation (specific meaning not provided in the original text).

**A.0.2 Detailed Explanation of Chi-Square Test** // The chi-square test helps us determine if the distribution of refactoring types in performance-related issues significantly differs from what we would expect based on general refactoring practices. To ensure the robustness of our statistical inference, we establish a significance level ($\alpha$) of 0.05. This threshold is widely accepted in scientific research and allows us to maintain a balance between Type I errors (false positives) and Type II errors (false negatives). Consequently, we will reject the null hypothesis and conclude that there is a statistically significant difference in the distribution of refactoring types between performance-related and non-performance-related refactorings if the p-value obtained from the Chi-Square test is less than 0.05. This null hypothesis posits that any observed differences in the distribution of refactoring types between performance-related and non-performance-related refactorings are due to chance rather than a systematic difference.

The variables in the chi-square test equation (Equation 3.1) are defined as follows:

- $\chi^2$: The chi-square statistic, measuring the overall difference between observed and expected frequencies.

- $O_i$: The observed frequency of the $i$-th refactoring type in performance-related refactorings.
- $E_i$: The expected frequency of the $i$-th refactoring type, assumed to be similar to those observed in non-performance-related refactorings for this analysis.

### A.0.3 Detailed Explanation of Cohen's D

Cohen's D is an effect size measure that quantifies the standardized difference between two means. In this context, it helps us understand the magnitude of the difference between performance-related and non-performance-related refactoring frequencies. Cohen's D is particularly valuable in this context as it goes beyond mere statistical significance to indicate the magnitude of the difference between the two groups. The primary objectives of this additional analysis are: (1) To provide deeper insights into the relative frequency of performance-related refactorings compared to non-performance-related refactorings. (2) To quantify the practical significance of the observed differences in the implementation of these refactoring types, complementing the statistical significance already established through the Chi-Square tests. (3) To establish a standardized measure of the difference between the two groups, allowing for meaningful comparisons across different refactoring types and potentially across the projects. The interpretation of Cohen's d values typically follows these guidelines[96]: (1) $|d| < 0.2$: Negligible effect (2) $0.2 \leq |d| < 0.5$: Small effect (3) $0.5 \leq |d| < 0.8$: Medium effect (4) $|d| \geq 0.8$: Large effect.

This analysis aims to bridge the gap between statistical significance and practical relevance, offering software engineers and researchers a more holistic understanding of how performance considerations influence refactoring decisions and practices. By quantifying the effect size, we can better inform decision-making processes in software development and maintenance, potentially leading to more targeted and effective performance issue resolution strategies.

The variables in the Cohen's d equation (Equation 3.9) are defined as follows:

- $M_1$: Mean frequency of non-performance-related refactorings across all projects.
- $M_2$: Mean frequency of performance-related refactorings across all projects.
- $SD$: Pooled standard deviation calculated across both refactoring groups.

### A.0.4 Detailed Explanation of Apriori Algorithm

The Apriori algorithm works on the principle that if an itemset is frequent, then all of its subsets must also be frequent. This property allows for efficient pruning of the search space,

making it feasible to analyze large datasets like ours. We iterate through the data, progressively identifying larger itemsets that meet a predefined support threshold. The output of this step is a set of frequently occurring combinations of refactoring types, which forms the basis for our subsequent analysis.

### A.0.5 Detailed Explanation of Association Rule Mining

Association rules provide insights into the co-occurrence of refactoring types, revealing how developers collaboratively apply them. We determine the appropriate threshold values for support and confidence through an iterative process. For support, we experiment with different minimum values to control itemset frequency, while confidence thresholds are adjusted to strengthen the identified associations. After fine-tuning, we set the support at 0.03 and confidence at 0.3. These thresholds are chosen based on dataset size and the need for meaningful, actionable patterns. A 0.03 support level ensures that associations appear in at least 3% of transactions, capturing significant patterns. A 0.3 confidence level prioritizes rules where the likelihood of one refactoring type following another is at least 30%, ensuring reliable insights for performance-related refactoring strategies.

### A.0.6 Detailed Explanation of Mann-Whitney U Test

The test statistic $U$ is computed as:

$$U = R_1 - \frac{n_1(n_1 + 1)}{2}$$

*where $R_1$ is the sum of ranks for one of the groups, and $n_1$ is the number of observations in that group. The $U$ statistic is then compared against critical values or converted into a p-value to assess statistical significance.*

### A.0.7 Detailed Explanation of Cliff's Delta Effect Size

It is defined as:

$$\Delta = \frac{(n_1 \cdot n_2) + \sum_{x \in X} \sum_{y \in Y} \text{sign}(x - y)}{n_1 \cdot n_2}$$

*where $n_1$ and $n_2$ are the sizes of the two groups, $X$ and $Y$ are the two datasets, and sign$(x - y)$ returns $1$, $-1$, or $0$ depending on whether $x > y$, $x < y$, or $x = y$, respectively. The resulting value, $\Delta$, is interpreted using thresholds: negligible ($|\Delta| < 0.147$), small ($0.147 \leq |\Delta| < 0.33$), medium ($0.33 \leq |\Delta| < 0.474$), and large ($|\Delta| \geq 0.474$).*

### A.0.8 Detailed Explanation of Odds Ratio

The odds ratio ($OR$) is calculated using the following formula:

$$OR = \frac{\frac{a}{b}}{\frac{(N_a - a)}{(N_b - b)}}$$

Where:

- $a$: Count of occurrences in the performance dataset for a specific refactoring type.
- $b$: Count of occurrences in the refactoring dataset for the same refactoring type.
- $N_a$: Total occurrences in the performance dataset (sum of all performance counts).
- $N_b$: Total occurrences in the refactoring dataset (sum of all refactoring counts).

**Explanation**

1. **Odds in the Performance Dataset ($\frac{a}{b}$):** This represents the ratio of the specific refactoring type's occurrences to the total occurrences of all types in the refactoring dataset.
2. **Odds in the Refactoring Dataset ($\frac{(N_a - a)}{(N_b - b)}$):** This calculates the complementary odds—how much the other types dominate the dataset compared to the specific type being analyzed.
3. **Odds Ratio ($OR$):** This is the ratio of the odds in the performance dataset to the odds in the refactoring dataset. It quantifies the likelihood of a specific refactoring type occurring in the performance dataset compared to its occurrence in the overall refactoring dataset.

# APPENDIX B DETAILED EXPLANATIONS AND CODE SNIPPETS FOR OUR ANALYSIS OF THE MAGNITUDE OF IMPACT OF VARIOUS REFACTORINGS TYPE ON PERFORMANCE CHANGE

### B.0.1 Scenarios where performance improved after refactoring:

1. **Extract Method:**

   - In the commit 80f912a [1], the `quickStart` method in the `HttpParser` class was refactored to enhance HTTP parsing efficiency by extracting specific parsing logic into a separate method.

```
// Before Refactoring
private void quickStart(ByteBuffer buffer) {
    // Existing parsing logic directly within this method
    // ...
}

// After Refactoring
private void quickStart(ByteBuffer buffer) {
    // Delegates specific parsing tasks to a new method
    parseRequestLine(buffer);
    // ...
}

private void parseRequestLine(ByteBuffer buffer) {
    // Extracted parsing logic for the request line
    // ...
}
```

   Although extracting the request line parsing logic from the `quickStart` method into `parseRequestLine` improved performance, the gains were minimal for several key reasons: the code path isn't frequently executed, the extracted functionality is narrow in scope, and the change primarily focused on maintainability rather

[1] https://github.com/eclipse/jetty.project/commit/80f912a121855749dc42c7866b03ed75f75cc9aa

than optimization. While the performance improvement was small, combining it with better code organization made the refactoring worthwhile.

2. **Extract Variable:**

- The commit 2af769f [2] in the *Netty* project involves enhancements to the `ByteBufUtil` class, particularly the `writeUtf8` method. The changes include adding overloads that allow writing UTF-8 encoded subsequences of `CharSequence` directly to a `ByteBuf` without creating temporary objects. This refactoring led to a significant performance improvement.

```
// Before Refactoring
public static int writeUtf8(ByteBuf buf, CharSequence seq) {
    // Original implementation
    // ...
    for (int i = 0; i < seq.length(); i++) {
        char c = seq.charAt(i);
        // Encoding logic
        // ...
    }
    // ...
}

// After Refactoring
public static int writeUtf8(ByteBuf buf, CharSequence seq) {
    // Refactored implementation with extracted variable
    int length = seq.length();
    // ...
    for (int i = 0; i < length; i++) {
        char c = seq.charAt(i);
        // Encoding logic
        // ...
    }
    // ...
}
```

The significant performance improvement from extracting `seq.length()` into a local variable in the `writeUtf8` method stemmed from several factors. Since this

[2] https://github.com/netty/netty/commit/2af769f6dc76e3aad88a52c345689d739a9130a2

method exists in a hot path of *Netty's* I/O handling, it's called frequently, making any optimization multiply across many invocations. By eliminating repeated method calls, the change reduced overhead and enabled better JVM optimization in tight loops [91, 97]. Additionally, the refactoring improved code clarity, demonstrating how seemingly minor changes to frequently executed code can yield substantial performance gains.

- In the commit 83c6981 [3] in the *Elastic APM Java Agent* project, the `currentTransaction()` method in the `ElasticApmTracer` class was refactored to enhance performance by avoiding unnecessary iterations when the active stack is empty.

```
// Before Refactoring
@Nullable
public Transaction currentTransaction() {
    final TraceContextHolder<?> bottomOfStack =
    ↪  activeStack.get().peekLast();
    if (bottomOfStack instanceof Transaction) {
        return (Transaction) bottomOfStack;
    } else {
        for (Iterator<TraceContextHolder<?>> it =
        ↪  activeStack.get().descendingIterator(); it.hasNext(); ) {
            TraceContextHolder<?> context = it.next();
            if (context instanceof Transaction) {
                return (Transaction) context;
            }
        }
    }
    return null;
}

// After Refactoring
@Nullable
public Transaction currentTransaction() {
    Deque<TraceContextHolder<?>> stack = activeStack.get();
    final TraceContextHolder<?> bottomOfStack = stack.peekLast();
    if (bottomOfStack instanceof Transaction) {
        return (Transaction) bottomOfStack;
```

[3] https://github.com/elastic/apm-agent-java/commit/83c698172889d68b49e25da02e9f4693ae8e367e

```
    } else if (bottomOfStack != null) {
        for (Iterator<TraceContextHolder<?>> it =
        ↪ stack.descendingIterator(); it.hasNext(); ) {
            TraceContextHolder<?> context = it.next();
            if (context instanceof Transaction) {
                return (Transaction) context;
            }
        }
    }
    return null;
}
```

The code was optimized by adding a null check for `bottomOfStack` to skip processing empty stacks. While this change improved performance by avoiding unnecessary iterations, particularly when `currentTransaction()` is frequently called on empty stacks, the gains were minimal for two reasons: empty stack checks already had minimal overhead, and the optimization's impact is limited to specific scenarios where empty stack processing is common.

3. **Change Variable Type:**

- In the commit c41d461 [4] in the *Netty* project, the `allocateRun` method in the `PoolChunk<T>` class was refactored to enhance performance by changing the data structures used for managing memory allocations.

```
//Before Refactoring
private long allocateRun(int runSize) {
    Long handle = runsAvail.pollFirst();
    // ... existing allocation logic ...
}

// After Refactoring
private long allocateRun(int runSize) {
    long handle = runsAvailHeap.poll();
    // ... updated allocation logic ...
}
```

[4] https://github.com/netty/netty/commit/c41d46111dc37aaf9c8ee7aec162d87221df1d70

The code was optimized by changing `Long` objects to primitive `long` values in the `runsAvailHeap`. This improves performance by eliminating boxing/unboxing operations and reducing memory overhead resulting in a minimal performance gains. Using custom implementations of priority queue also helped optimize memory allocation.

- In the commit 09a441f [5] in the *Zipkin* project, the `Proto3Codec.readList` method was refactored to enhance performance by changing the variable type used for buffer management.

```
// Before Refactoring
public static boolean readList(byte[] bytes, Collection<Span> out) {
    Buffer buffer = new Buffer(bytes);
    // ... existing parsing logic ...
}

// After Refactoring
public static boolean readList(byte[] bytes, Collection<Span> out) {
    UnsafeBuffer buffer = new UnsafeBuffer(bytes);
    // ... updated parsing logic ...
}
```

While switching from `Buffer` to `UnsafeBuffer` improved performance by reducing bounds checks and enabling faster memory access, the gains were minimal for two reasons: the parsing logic wasn't frequently executed enough to maximize the benefits, and the performance advantage from bypassing safety checks had limited impact on overall system metrics. This change demonstrates how selecting specialized data structures can enhance efficiency in performance-critical applications, particularly in systems like *Zipkin* that require rapid processing of tracing data.

4. **Add Variable Modifier:**

- In the commit d985e93 [6] in the *Zipkin* project, the `build` method in the `BulkCallBuilder` class was refactored to enhance performance and code clarity by changing variable types and adding the `final` modifier.

[5] https://github.com/openzipkin/zipkin/commit/09a441f4b4e8734058e8d023367f826b845e9dea
[6] https://github.com/openzipkin/zipkin/commit/d985e932aa8b700b82c39859adbc4b3ef2c0ca1b

```java
// Before Refactoring
public HttpCall<Void> build() {
    Map<String, String> headers = new LinkedHashMap<>();
    // ... existing logic ...
}

// After Refactoring
public HttpCall<Void> build() {
    final Map<String, String> headers = new LinkedHashMap<>();
    // ... updated logic ...
}
```

Although adding the `final` modifier to a local variable enables some compiler optimizations, the performance impact was minimal because the change was limited in scope, affected a relatively simple computation, and primarily served to improve code clarity rather than optimize performance.

5. **Inline Variable:**

- The commit c46b9bb [7] in the *Jdbi* project involves refactoring the `GenericTypes` class, particularly the `findGenericParameter` method. This change replaces using Guava's reflection utilities with the `GeAnTyRef` library, resulting in performance improvements.

```java
// Before Refactoring
public static Optional<Type> findGenericParameter(Type type, Class<?>
↪  parameterizedSupertype, int n) {
    // Original implementation using Guava's TypeToken
    TypeToken<?> token = TypeToken.of(type);
    TypeToken<?> supertype =
    ↪  token.getSupertype(parameterizedSupertype);
    Type resolvedType = supertype.getType();
    // Further processing to extract the nth generic parameter
    // ...
    return Optional.ofNullable(resolvedParameter);
}
```

[7] https://github.com/jdbi/jdbi/commit/c46b9bbd82fab5f7caa0c5cec5a8457a889a1222

```
//After Refactoring
public static Optional<Type> findGenericParameter(Type type, Class<?>
↪  parameterizedSupertype, int n) {
    // Refactored implementation using GeAnTyRef
    Type resolvedType = GenericTypeReflector.getExactSuperType(type,
    ↪  parameterizedSupertype);
    // Further processing to extract the nth generic parameter
    // ...
    return Optional.ofNullable(resolvedParameter);
}
```

The significant performance improvement from replacing Guava's `TypeToken` with `GeAnTyRef's` utilities in *Jdbi's* `findGenericParameter` method stemmed from three factors: `GeAnTyRef's` more focused approach reduced library overhead, direct type resolution with `GenericTypeReflector.getExactSuperType` streamlined processing and fewer intermediate objects decreased memory usage.

6. **Extract Method + Remove Method Annotation:**

- In the commit aad0b60 [8] in the *Jdbi* project, the `AbstractArgumentFactory` class underwent made two main changes. To improve code organisation, it extracted part of the build method's logic into a new private `innerBuild` method. Additionally, it removed the `@Override` annotation from the `build` method, indicating it no longer overrides a parent class method.

```
// Before Refactoring
@Override
public Optional<Argument> build(Type type, Object value, ConfigRegistry
↪  config) {
    // Original implementation logic
}

// After Refactoring
public Optional<Argument> build(Type type, Object value, ConfigRegistry
↪  config) {
    return innerBuild(value, config)
```

[8] https://github.com/jdbi/jdbi/commit/aad0b605631763fa4262f43a960fc8c92f9ecc4f

```
        .orElseThrow(() -> new
        ↪  UnableToCreateStatementException("Prepared argument " +
        ↪  value + " of type " + type + " failed to bind"));
}

private Optional<Argument> innerBuild(Object value, ConfigRegistry
↪  config) {
    // Extracted core logic from the original build method
}
```

While extracting logic into the `innerBuild` method and removing the `@Override` annotation enhanced code clarity, the performance impact was minimal for several reasons: the code is executed with the same frequency, the changes focused on maintainability rather than optimization, and the underlying algorithm remained unchanged.

7. **Extract Variable + Change Variable Type + Inline Variable:**

- In the commit d6f623b [9] in the *Fastjson2* project, the `createObjectWriter` method in the `ObjectWriterCreatorASM` class was refactored to enhance performance and maintainability by extracting variables, changing variable types, and inlining variables.

```
// Before Refactoring
public ObjectWriter createObjectWriter(Class objectClass, long
↪  features, ObjectWriterProvider provider) {
    // ... existing logic ...
    String className = objectClass.getName();
    // ... existing logic ...
}

//After Refactoring
public ObjectWriter createObjectWriter(Class objectClass, long
↪  features, ObjectWriterProvider provider) {
    // ... existing logic ...
    String className = TypeUtils.getTypeName(objectClass);
```

[9] https://github.com/alibaba/fastjson2/commit/d6f623b900fc10847f381c80cc09f08ad40246d0

```
    // ... updated logic ...
}
```

While switching from `objectClass.getName()` to `TypeUtils.getTypeName(objectClass)` centralized the class name retrieval logic, the performance impact was minimal because the change affected a narrow scope, optimization benefits depended heavily on usage patterns, and the refactoring primarily served to improve maintainability rather than performance.

### B.0.2 Scenarios where performance regressed after refactoring:

1. **Extract Method:**

   - In the commit bc8b4b1 [10] in the *Feign* project, the `proceed()` method in the `InvocationContext` class was refactored to enhance code clarity and maintainability by extracting a portion of its logic into a separate method but resulted in a minimal performance regression.

```
// Before Refactoring
public Object proceed() throws Throwable {
    // Original logic handling the invocation
    // ...
    // Directly processing the response
    if (response.status() >= 200 && response.status() < 300) {
        // Success logic
    } else {
        // Error handling logic
    }
    // ...
}

//After Refactoring
public Object proceed() throws Throwable {
    // Original logic handling the invocation
    // ...
```

[10] https://github.com/OpenFeign/feign/commit/bc8b4b1fc53f0fc0911f486a5150fedab6813798

```
    // Delegating response processing to a new method
    return handleResponse(response);
}

private Object handleResponse(Response response) throws Throwable {
    if (response.status() >= 200 && response.status() < 300) {
        // Success logic
    } else {
        // Error handling logic
    }
}
```

Moving the response-handling logic to a separate `handleResponse` method made the code more readable and modular. While this introduced minor performance overhead from the extra method call, particularly in high-throughput cases where JVM inlining might not occur, the enhanced maintainability generally justifies this small performance regression. However, performance-critical systems should carefully evaluate the impact of such refactoring decisions

2. **Extract Variable:**

- The commit de13cef [11] in the *Jetty* project involves modifications to the `Pool` class, particularly the reserve method. The changes include extracting variables to enhance code readability and maintainability. However, this refactoring resulted in a significant performance regression.

```
// Before Refactoring
public Entry<T> reserve() {
    // Original implementation with inlined logic
    if (head == null) {
        // Handle empty pool
        return null;
    }
    Entry<T> entry = head;
    head = entry.next;
    entry.next = null;
```

[11] https://github.com/eclipse/jetty.project/commit/de13ceff3619777dc1213a4dddc40d79ceedb3dd

```
    return entry;
}

// After Refactoring
public Entry<T> reserve() {
    // Extracted variables for clarity
    final Entry<T> firstEntry = head;
    if (firstEntry == null) {
        // Handle empty pool
        return null;
    }
    final Entry<T> nextEntry = firstEntry.next;
    head = nextEntry;
    firstEntry.next = null;
    return firstEntry;
}
```

The refactoring of the `Pool.reserve()` method improved code readability by adding new variables (`firstEntry, nextEntry`), but introduced performance regression. While these additional variables make the code more maintainable, they create overhead through increased allocation rates and potential disruption of JIT compiler optimizations. Since `reserve()` is frequently called, these small overheads can significantly impact performance, particularly in high-throughput scenarios or performance-sensitive applications where garbage collection pressure and JIT optimization matter.

- In the commit a9e631a [12] in the *Feign*, the `processAnnotationOnClass` method in the `DeclarativeContract` class was refactored to enhance code clarity by extracting variables which resulted in a minimal performance regression.

```
// Before Refactoring
protected final void processAnnotationOnClass(MethodMetadata data,
↪  Class<?> targetType) {
    // Existing logic with inline expressions
    if (targetType.isAnnotationPresent(SomeAnnotation.class)) {

        ↪  data.addAnnotation(targetType.getAnnotation(SomeAnnotation.class));
```

[12] https://github.com/OpenFeign/feign/commit/a9e631ac9beed8a59b52b480f75231695624c2a0

```
    }
    // Additional processing...
}

// After Refactoring
protected final void processAnnotationOnClass(MethodMetadata data,
↪  Class<?> targetType) {
    // Extracted variable for clarity
    SomeAnnotation annotation =
    ↪  targetType.getAnnotation(SomeAnnotation.class);
    if (annotation != null) {
        data.addAnnotation(annotation);
    }
    // Additional processing...
}
```

The refactoring of the `DeclarativeContract.processAnnotationOnClass` method introduced a new annotation variable that improved code clarity and maintainability. While this change adds minimal overhead in memory and assignment costs, its impact is minimal since modern JVMs optimize such assignments and the variable is short-lived. The enhanced readability and easier maintenance outweigh these minor performance regressions in typical usage scenarios.

3. **Change Variable Type:**

- In commit 246df37 [13] in the *Feign* project, the `target` method `RequestTemplate` class was refactored to modify how URI paths are handled, particularly concerning the addition of leading slashes which resulted in a minimal performance regression.

```
// Before Refactoring
public RequestTemplate target(String target) {
    // Existing logic
    if (target.endsWith("/")) {
        target = target.substring(0, target.length() - 1);
    }
    // Additional processing...
```

[13] https://github.com/OpenFeign/feign/commit/246df3746ac3af05945a170940ff7e78a97ce038

```
}

//After Refactoring
public RequestTemplate target(String target) {
    // Updated logic
    if (target.endsWith("/")) {
        target = target.substring(0, target.length() - 1);
    }
    if (!target.startsWith("/")) {
        target = "/" + target;
    }
    // Additional processing...
}
```

The refactoring standardized URI formatting by ensuring leading slashes, which prevents routing errors and malformed URIs. While this introduced minor regression in performance from string operations (checking, substring, concatenation), these costs are negligible in practice. The improved reliability and maintainability justify this minimal performance trade-off.

4. **Inline Variable:**

- The commit 750584b [14] in the *Jetty* project involves modifications to the `ArrayByteBufferPool` class, particularly the `reserve` method. The changes include inlining variables and adjusting memory management strategies, which resulted in a significant performance regression.

```
// Before Refactoring
private void reserve(RetainedBucket bucket, RetainableByteBuffer
↪  buffer) {
    int size = buffer.getSize();
    // Logic utilizing the 'size' variable
    // ...
}

// After Refactoring
```

[14] https://github.com/eclipse/jetty.project/commit/750584bc853d9ba300f8d3fb2433158f0968ebbf

```
private void reserve(RetainedBucket bucket, RetainableByteBuffer
↪  buffer) {
    // Inlined the 'size' variable directly into the logic
    // ...
    if (buffer.getSize() > threshold) {
        // Logic that previously used the 'size' variable
        // ...
    }
    // ...
}
```

Removing a cached variable for `buffer.getSize()` led to performance degradation due to repeated method calls, which is especially impactful if the method involves non-trivial work or synchronization. This change also made the code less readable and may have prevented JVM optimizations like inlining. The impact is particularly significant since this occurs in a performance-critical path, demonstrating how seemingly small changes can substantially affect execution when code clarity is sacrificed for perceived optimization.

- In the commit 0eac0af [15] in the *Elastic APM Java Agent* project, the `currentTransaction()` method in the `ElasticApmTracer` class underwent a refactoring that involved inlining a variable to streamline the code which resulted in a minimal performance regression.

```
// Before Refactoring
public Transaction currentTransaction() {
    Transaction transaction = getActiveTransaction();
    return transaction != null ? transaction :
    ↪  NoopTransaction.INSTANCE;
}

// After Refactoring
public Transaction currentTransaction() {
    return getActiveTransaction() != null ? getActiveTransaction() :
    ↪  NoopTransaction.INSTANCE;
}
```

[15] https://github.com/elastic/apm-agent-java/commit/0eac0af814b8686772c6d4e2212f4231524fa8ce

The removal of an intermediate variable made the code more concise but introduced multiple calls to `getActiveTransaction()` in a single `return` statement. While this might seem minor, it can cause performance regression if the method involves more than simple field access, particularly in high-throughput scenarios. Though modern JVMs can optimize such cases and the impact is usually small, the change demonstrates how pursuing code brevity can inadvertently introduce performance costs.

5. **Extract Method + Change Variable Type:**

- The commit fd9da3e [16] in the *Prometheus Java client* project involves refactoring the `CKMSQuantiles` class, particularly the `get(double q)` method. The changes include extracting methods and altering variable types to enhance code readability and maintainability. However, this refactoring resulted in a performance regression.

```
// Before Refactoring
public double get(double q) {
    // Original implementation with inlined logic
    List<Item> samples = ...; // Assume this is a List
    // Logic to compute quantile 'q' using 'samples'
    // ...
}

// After Refactoring
public double get(double q) {
    // Refactored implementation with extracted method and changed
    ↪ variable type
    Collection<Item> samples = getSamples();
    return computeQuantile(q, samples);
}

private Collection<Item> getSamples() {
    // Logic to retrieve samples, now returning a Collection instead of
    ↪ a List
    // ...
}
```

[16] https://github.com/prometheus/client_java/commit/fd9da3e7f756dc9c119108ebf6dbe88cda9a740c

```
private double computeQuantile(double q, Collection<Item> samples) {
    // Logic to compute quantile 'q' using 'samples'
    // ...
}
```

This refactoring changed the data structure from `List` to `Collection`, which sacrificed constant-time random access efficiency since Collection may require iteration instead. While this change improved code modularity, it introduced performance overhead through additional method calls and potentially less efficient collection handling, especially if implemented with non-array-backed structures. This illustrates how architectural changes can significantly impact performance in statistical computation methods even when they enhance code organization.

6. **Extract Method + Extract Variable:**

- The commit 6ef7785 [17] in the *SimpleFlatMapper* project involves refactoring the `matches` method within the `PropertyMatchingScore` class. The refactoring includes method extraction and variable extraction and results in a performance regression.

```
// Before Refactoring
public PropertyMatchingScore matches(String property) {
    // Original implementation with inlined logic
    // ...
}

// After Refactoring
public PropertyMatchingScore matches(String property) {
    // Extracted method and variables
    String processedProperty = processProperty(property);
    return calculateMatchingScore(processedProperty);
}

private String processProperty(String property) {
    // Logic extracted from matches method
```

[17] https://github.com/arnaudroger/SimpleFlatMapper/commit/6ef7785ddd157a014f79af09ab8baa750a8e0049

```
    // ...
}

private PropertyMatchingScore calculateMatchingScore(String property) {
    // Logic extracted from matches method
    // ...
}
```

This refactoring introduced performance overhead by extracting functionality into separate methods (`processProperty` and `calculateMatchingScore`). This added method call costs and potentially prevented JVM optimizations like inlining, especially problematic since `matches` is frequently called. The restructuring may have also disrupted JVM caching patterns, leading to less efficient execution in this performance-critical path [98].

### B.0.3 Scenarios where performance unchanged after refactoring:

1. **Extract Method:**

   - In commit 2299c0b [18] in the project *Jersey*, the `configure` method in the `ExternalPropertiesCo` class was refactored to enhance code clarity and maintainability by extracting certain functionalities into separate methods.

```
// Before Refactoring
public static boolean configure(Configurable config) {
    // Complex logic handling configuration
    // ...
    // Directly processing properties
    if (properties != null) {
        for (Map.Entry<String, String> entry : properties.entrySet()) {
            config.property(entry.getKey(), entry.getValue());
        }
    }
    // ...
}
```

[18] https://github.com/eclipse-ee4j/jersey/commit/2299c0bd7e32420b08857a2aba16dd77dd13f7a1

```
// After Refactoring
public static boolean configure(Configurable config) {
    // Delegating property processing to a new method
    Map<String, String> properties = loadExternalProperties();
    applyProperties(config, properties);
    // ...
}

private static void applyProperties(Configurable config, Map<String,
↪  String> properties) {
    if (properties != null) {
        for (Map.Entry<String, String> entry : properties.entrySet()) {
            config.property(entry.getKey(), entry.getValue());
        }
    }
}
```

The refactoring extracted property application logic into a new applyProperties() method to improve code organization and readability. Since this was purely a structural change that separated existing functionality, it had no impact on performance.

2. **Extract Variable:**

- In commit 4a6c274 [19] in the *Jersey*, the `newExchangeQueue` method in the `HttpDestination` class was refactored to enhance memory efficiency by modifying the initialization strategy of the request queue.

```
// Before Refactoring
protected Queue<HttpExchange> newExchangeQueue(HttpClient client) {
    return new
    ↪  BlockingArrayQueue<>(client.getMaxRequestsQueuedPerDestination());
}

// After Refactoring
protected Queue<HttpExchange> newExchangeQueue(HttpClient client) {
```

[19] https://github.com/eclipse/jetty.project/commit/4a6c2744a0d4ff404daee0a521e88ae386356e06

```
    int maxCapacity = client.getMaxRequestsQueuedPerDestination();
    int initialCapacity = Math.min(32, maxCapacity);
    return new BlockingArrayQueue<>(initialCapacity, initialCapacity,
    ↪  maxCapacity);
}
```

The refactoring implemented dynamic queue sizing by replacing fixed pre-allocation with on-demand growth logic. Since this was purely a memory optimization that modified allocation strategy, it had no impact on runtime performance