Empirical studies on code refactoring practices in software development process

By Shayan Noei

A thesis submitted to the Graduate Program in Department of Electrical and Computer Engineering in conformity with the requirements for the Degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
October, 2024

Abstract

Refactoring is a crucial aspect of code maintenance, involving modifications to the internal structure of code without altering its external behavior. Developers often invest significant effort in improving code through refactoring. Refactoring provides benefits, such as simplifying maintenance, enhancing code readability, and helping developers gain a better understanding of the code. Due to the absence of large-scale studies and comprehensive quality evaluations, the refactoring practices employed by developers and their impact on code quality during software development remain largely unexplored. For example, it is unclear how often developers engage in refactoring and whether frequent refactoring leads to better code quality compared to infrequent refactoring. Additionally, the lack of comprehensive refactoring tools for popular programming languages like Python, which is commonly used in emerging domains such as machine learning (ML), skews research toward more widely studied languages like Java, which has benefited from more advanced refactoring detection tools in recent years. This thesis addresses these gaps by conducting three empirical studies to (1) identify short-term and long-term refactoring strategies and their relationship with code quality, (2) analyze release-wise refactoring patterns and their impact on release quality, and (3) detect and study refactoring-related commits in

Python ML projects through an ML-based refactoring detection approach. The research conducted in this thesis helps practitioners understand current refactoring trends, their relationship with code quality, and provides a machine learning-based refactoring detection tool for studying refactoring practices in the Python ML domain, which is intended to guide future research toward less-studied languages, such as Python.

Co-Authorship

With this statement, I, Shayan Noei, hereby declare that the research presented in this thesis is my own work. All publications, ideas, and inventions from others have been properly cited.

The publications related to this thesis are listed as follows:

- Noei, S., Li, H., Georgiou, S., Zou, Y. (2023). **An Empirical Study of Refactoring Rhythms and Tactics in the Software Development Process. IEEE Transactions on Software Engineering**, 49(12), 5103-5119.

I am the primary author of the publication mentioned above (Chapter 3), co-authored with Dr. Ying Zou, Dr. Heng Li, and Dr. Stefanos Georgiou. Dr. Ying Zou supervised the research, while the co-authors participated in meetings and provided me feedback and suggestions to enhance my work.

Acknowledgments

First of all, I want to thank God for granting me the ability and strength to accomplish this thesis. I want to express my gratitude to my supervisor, Dr. Ying (Jenny) Zou, for her invaluable contributions to my growth both as a researcher and as an individual. Dr. Zou has fostered an environment that has significantly shaped my development into the researcher I am today. She has consistently made herself available to assist with my research progress at any time. I would also like to extend my gratitude to Dr. Heng Li for his support and collaboration throughout this thesis. His insights and shared experiences have been invaluable; without his ideas and guidance, completing this thesis would have been much more challenging. A special thanks to my brother, Dr. Ehsan Noei, who has been consistently supportive throughout my studies. His mentorship has been crucial in reaching this milestone. Lastly, I am deeply grateful to my parents for their unwavering support of my academic and personal decisions. Their availability and encouragement whenever I needed them have been essential to my accomplishments. I appreciate my examining committee, Dr. Bram Adam, Dr. Xiandan Zhu, Dr. Ali Ouni, and Dr. Michael Korenberg for their time and valuable feedback, which helped me to improve my thesis. I had the privilege of working with an exceptional group of researchers in the software evolution and analytics lab (SEAL) Lab: Dr. Stefanos Georgiou, Dr. Taher A. Ghaleb, Dr.

Ossama Ehsan, Dr. Guoliang Zhao, Dr. Safwat Hassan, Dr. Maram Assi, Chunli Yu, Yiping Jia, Fangjian Lei, Pouya Fathollahzadeh, Jiawen Liu, Jonathan Cordeiro, Bihui Jin, and Omar El Zarif. Finally, I would like to thank all my friends; without their support, this thesis would not have been possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Maintaining high-quality software is essential for the success of software systems [1, 2]. Software quality is divided into internal and external quality aspects [3]. Internal quality focuses on technical and design elements, which can be directly measured through static analysis of the source code [4]. In contrast, external quality attributes, such as usability—referring to how easy the system is to use—relate to meeting user needs, necessitating user involvement. Refactoring is a systematic process aimed at improving the internal quality of software without altering its external functionalities [5]. Refactoring enhances the extensibility and maintainability of a software system [6–8], which can be motivated by various factors, such as improving software design, reducing maintenance effort, and eliminating code smells [9–13]. As a result, developers often invest substantial effort in refactoring as part of the software maintenance process [14, 15]. Developers utilize different practices to address refactoring tasks [16, 17]. However, due to the lack of large-scale empirical studies, there is limited understanding of common refactoring practices across different development cycles and their impact on code quality. This includes how refactoring is performed on weekdays (*i.e.,* refactoring rhythms), over the long term (*i.e.,* refactoring tactics),

and within each release cycle (*i.e.,* release-wise refactoring patterns), as well as which specific practices contribute to higher quality code. Understanding these practices is crucial for determining the appropriate timing for development, facilitating software evolution, making informed decisions for code changes, and deepening knowledge of code design [18–20].

Various tools, such as Rminer [10, 21], have been developed to detect refactoring operations in the history of Java code. These tools play a significant role in refactoring studies for specific languages like Java. Python has become one of the most popular programming languages, especially for developing Artificial Intelligence (AI) and Machine Learning (ML) libraries, frameworks, and applications [22–25]. However, due to different coding practices, refactoring detection tools for Python are relatively limited, leaving refactoring operations in domains such as ML, which largely use Python, largely unexplored. Therefore, developing effective refactoring detection techniques that overcome the limitations of current state-of-the-art methods—often restricted in their applicability to Python—is essential. This would enable practitioners to better study refactoring practices in Python and domains like ML that rely heavily on this language.

1.1 Background

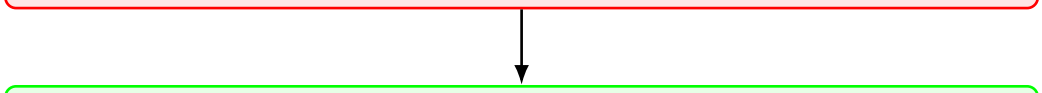
In this sub section, we provide a background on code refactoring, refactoring detection, refactoring practices, and code smells.

1.1.1 Code Refactoring

Refactoring involves a series of small transformations, known as refactoring operations or refactoring types, which preserve the software's behavior [26]. Each refactoring type (*e.g.,* rename an attribute) involves a specific operation that impacts certain levels of the code. For instance, refactoring types can range from simple operations, such as extracting a variable, to more extensive changes at the class level, such as pulling up a method, which involves moving a method from a subclass to a superclass to reduce code duplication and enhance reuse [27]. Although each refactoring type may have a minor effect, a sequence of these transformations can significantly restructure the code, reducing the likelihood of system breakage [26]. Figure 1.1 shows an example of the extract variable refactoring operation, which involves moving the discount amount calculation into a separate variable to enhance the readability and maintainability of the code.

Before Refactoring

```
def calculate_discounted_price(price, discount):
    return price - (price * discount / 100)
```



After Refactoring

```
def calculate_discounted_price(price, discount):
    discount_amount = price * discount / 100
    discounted_price = price - discount_amount
    return discounted_price
```

Figure 1.1: Example of the Extract Variable refactoring.

1.1.2 Refactoring Detection

As developers do not always provide sufficient documentation on their refactoring activities [20], refactoring detection refers to the automated process of identifying refactorings within the development history. This process helps developers to better understand the codebase [18], learn from past activities, and assist both developers and managers in recognizing previous refactoring efforts, thereby enhancing knowledge transfer. Consequently, it enables a more effective code review process [28] and facilitates better planning and resource allocation for future development.

Existing approaches for refactoring detection can be classified into two categories: rule-based and keyword-based. Rule-based tools, such as Rminer [10, 21], extract refactoring activities from the development history by performing static analysis of the source code. For instance, Rminer relies on pattern matching using common refactoring patterns in the Abstract Syntax Tree (AST) [10, 21] to detect refactoring operations. These tools are often evaluated based on their coverage of refactoring types (*i.e.,* refactoring operations) and their accuracy in detecting refactorings.

Keyword-based approaches detect refactoring-related commits by analyzing commit messages alone [29–32]. These methods do not rely on code structure or coding styles, making them language-independent and dependent exclusively on the presence of specific keywords, such as "refactor" in commit messages to indicate refactoring activities.

Among all refactoring detection tools, Rminer stands out for its comprehensive type coverage and accuracy, with an overall precision of 99.7% and a recall of 94.2% [10]. Rminer version 3.0.7 is capable of identifying up to 102 refactoring types in the history of a Java codebase [10, 21].

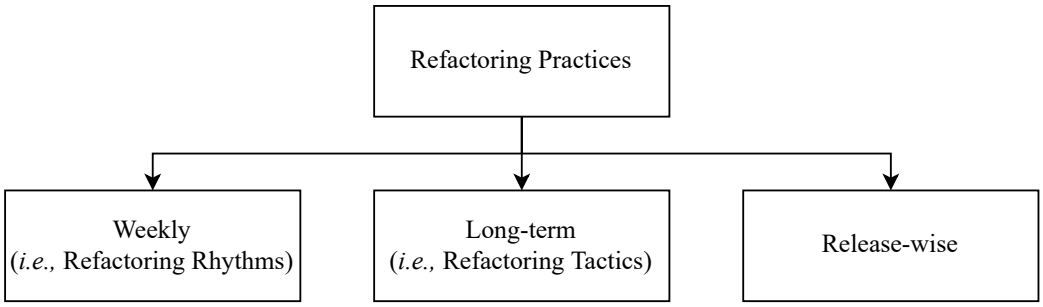


Figure 1.2: Refactoring practices studied in this thesis during the software development cycle.

1.1.3 Refactoring Practices

Due to the significant effort developers invest in refactoring [14, 15], they may adopt various practices that reflect how refactoring is integrated into the development process. Different teams might employ diverse refactoring strategies for both short-term and long-term periods [16, 17]. Therefore, refactoring practices are common strategies that developers or teams use to distribute refactoring tasks across various stages of the development lifecycle. Existing refactoring practices are broadly categorized into two types: floss and root canal refactorings. Floss refactoring involves frequent, incremental refactoring integrated into the regular development process, whereas Root canal refactoring involves occasional, more extensive refactoring performed separately from the routine development activities. However, due to the lack of large-scale empirical studies, the extent and side effects of these practices remain poorly understood. As illustrated in Figure 1.2, we categorize and study refactoring practices into three types: (1) weekly, (2) long-term, and (3) release-wise. Each type addresses a distinct aspect of software development:

- **Weekly:** Describe how refactoring activities are distributed across weekdays and weekends, referred to as refactoring rhythms throughout the rest of the

thesis.

- **Long-term:** The distribution of refactoring practices over extended periods (*e.g.,* two years of development), which is referred to as refactoring tactics in the rest of the thesis.

- **Release-wise:** The distribution of refactorings within each software release, is usually influenced by release dates.

1.1.4 Code Smells

Code smells are poor coding practices that indicate design flaws in the code [33] impairing the maintainability of software [34, 35]. For instance, a broken hierarchy code smell happens when a subtype and its supertype do not share an "IS-A" relationship [36]. Code smells can be categorized into different levels of granularity, including architecture smells, which relates to issues with the overall system structure and dependencies; design smells, which involve problems with fundamental design principles of code; testability smells, which address aspects related to the testability of the code; implementation smells, which are concerned with problems in code implementation; and test smells, which relate to issues within the test code [37]. Refactoring can be motivated by various reasons, such as improving software design [38], making software systems easier to understand [18], enhancing reusability [39], removing dependencies among attributes, methods, classes, interfaces, and packages [40], as well as eliminating code smells [9–13, 40]. Even though refactoring can be motivated by many reasons, it has been shown to result in the reduction or elimination of code smells [10–12, 41]. For example, Figure 1.3 illustrates how to resolve the magic number code smell, a type of implementation smell, through the extract variable refactoring type. A magic

Before Refactoring

```
def calculate_discounted_price(price):
    return price * 0.15
```

After Refactoring

```
DISCOUNT_RATE = 0.15

def calculate_discounted_price(price):
    return price * DISCOUNT_RATE
```

Figure 1.3: Example of the extract variable refactoring type, which eliminates the magic number code smell.

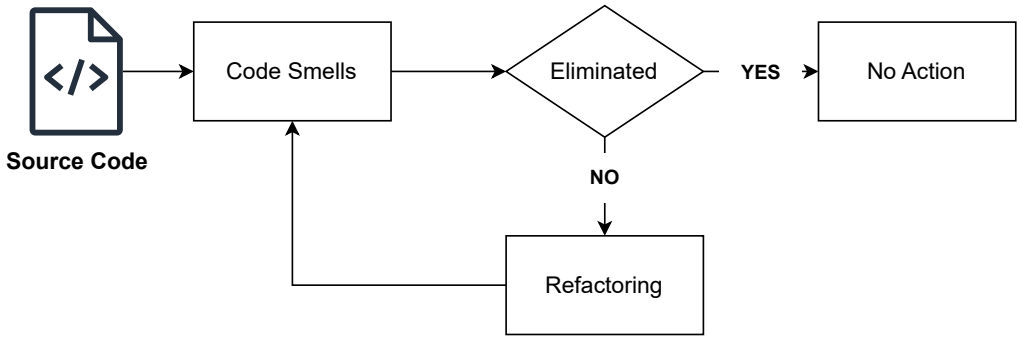


Figure 1.4: The process of eliminating code smells through refactoring.

number code smell refers to a literal number in the source code that lacks an obvious meaning [42], making the code harder to understand and maintain. Therefore, the success of refactoring is often evidenced by the reduction or elimination of code smells, which may require developers to perform multiple refactoring operations to remove them [13]. Figure 1.4, provides an overview of how code smells can be eliminated through refactoring.

1.2 Research Problems

Refactoring is a crucial part of the software maintenance process, yet there are limited studies analyzing refactoring practices in software development. The state-of-the-art [5,43,44] research relies on two abstract terms: floss and root canal refactoring, which refer to consistent refactoring alongside development and occasional, intensive refactoring activities, respectively. However, the lack of large-scale refactoring datasets, comprehensive refactoring detection tools, and robust analytical methods has left the identification of common refactoring practices across various development scopes largely unexplored. Additionally, the association between refactoring practices and code quality has not been analyzed in a fine-grained manner, with terms like high or low quality often lacking detailed analysis to convince practitioners of their impacts at different levels of the code. Furthermore, existing studies primarily focus on languages like Java, which are well supported by existing refactoring detection tools, while the absence of sophisticated tools for popular languages such as Python makes it difficult to investigate refactoring practices within Python projects.

These challenges prompt researchers to investigate common refactoring practices, and their relationship with code quality, and to extend research to other programming languages, such as Python. Thus, we envision the following research problems related to refactoring in the software development process:

Problem I: Developers are unaware of short-term and long-term refactoring strategies and their effects on code quality. Prior studies have categorized refactoring strategies using the abstract terms floss and root canal tactics [5, 43, 44]. However, the lack of large-scale studies has made it difficult to determine how developers integrate these patterns into software development and,

more importantly, how they impact code quality. This issue is due to the absence of a large-scale refactoring dataset, which limits the generalizability of findings from smaller datasets that may suffer from contextual variability. To address this, it is crucial to examine a diverse range of projects with varying external factors, such as team size, to accurately identify common refactoring patterns. Moreover, simply associating each pattern with low or high quality is insufficient to convince practitioners of the benefits of each refactoring pattern. To effectively demonstrate these benefits, it is essential to provide fine-grained results. This requires extensive data and a detailed analysis of refactoring strategies. A comprehensive dataset, coupled with thorough analysis, will offer a clearer understanding of how different patterns influence code quality, thereby supporting more informed decisions by practitioners. Therefore, we aim to collect a large-scale refactoring dataset and study refactoring strategies from both short-term and long-term perspectives. Our goal is to provide a common vocabulary for practitioners and offer detailed best practices for refactoring throughout the software development cycle. For short-term strategies, we will investigate how developers integrate refactoring activities into their weekly tasks (*i.e.,* refactoring rhythms) and examine their relationship with code quality. For long-term practices (*i.e.,* refactoring tactics), we aim to identify additional tactics or variations of existing approaches like floss and root canal, and assess their correlation with code quality.

Problem II: The integration of refactoring within release cycles is not well understood, making it difficult to identify the most effective approaches. With the adoption of continuous integration and continuous development

(CI/CD) [45], software projects now release new versions in shorter cycles, automatically and continuously [46, 47]. These shorter release cycles enable developers to effectively leverage user feedback [48] and promptly deliver updates or fixes based on that feedback. As a result, fast-release cycles can range from just a few days to weeks [49–51], whereas traditional software development often takes years to publish a new version [49, 52]. Given the substantial effort developers invest in refactoring [14, 15], it remains unclear how these efforts are distributed throughout release cycles. Identification of the common refactoring practices within releases is challenging due to the diverse release practices and branch strategies that developers use when releasing new features, such as the main branch release strategy, which makes an analysis based solely on individual commits within releases potentially inaccurate, as releases may occur from specific branches rather than all branches. Additionally, external factors such as hot fixes or feature deliveries introduce noise, complicating the identification of common refactoring practices. Furthermore, studying refactoring and establishing generalizable and fine-grained relationships with code quality within release cycles requires a large-scale dataset from projects implementing CI/CD processes, which is currently lacking. We aim to address this gap by first gathering a large-scale dataset of projects that utilize CI/CD processes, detecting their refactoring activities, and then analyzing refactoring strategies within short-term release cycles to identify patterns that contribute to high-quality software delivery.

Problem III: Limited tools for refactoring detection in Python restrict studies on language-specific and domain-specific refactoring practices. Python has become one of the most popular programming languages, particularly for developing machine-learning frameworks and libraries [22–25], making it crucial to study

and analyze refactoring practices within this language. However, Python has limited tooling for refactoring detection, largely due to its unique development practices compared to other programming languages. Python prioritizes readability [53, 54] and has a simpler coding structure [54, 55], and as a dynamically typed language [54], Therefore, implementing state-of-the-art refactoring detection methods in Python, which primarily rely on pattern matching through abstract syntax trees (ASTs) and the identification of class and object relationships through static code analysis, is more challenging. Additionally, the lack of domain-specific studies makes it unclear how refactoring practices evolve in emerging fields, such as ML, where data-driven architectures may require data-specific refactoring operations. Consequently, novel approaches are required to detect refactorings when traditional code analysis techniques are insufficient. Current Python refactoring detection tools (*i.e.,* PyRef [19]), cover 11% of the refactoring operations detectable by RMiner [10, 21] for Java. This limited coverage leaves numerous refactorings undetected, making the analysis of refactoring practices challenging. Therefore, we aim to develop a machine-learning-based refactoring detection tool by collecting a large-scale dataset of refactorings from Python ML projects and analyzing how refactoring practices vary within the ML technical domain.

1.3 Thesis Statement

Refactoring is a critical aspect of the software maintenance process, requiring developers to invest significant effort in improving source code. However, the lack of large-scale datasets raises concerns about the generalizability and context variability

of the refactoring tactics used in prior studies (*i.e.,* floss and root canal). Additionally, their detailed correlation with various aspects of code quality remains unclear, leading to uncertainty among practitioners about their side effects and effectiveness at different levels of the code. Moreover, the lack of comprehensive refactoring detection tools for trending programming languages, such as Python, has resulted in limited evidence and understanding of refactoring practices within the software development process, particularly in languages and domains that predominantly use these languages (*e.g.,* the ML domain in Python).

This thesis aims to explore extended refactoring practices across the software development lifecycle and measure their relationship with code quality. We present two large-scale refactoring datasets for Java and one large-scale refactoring dataset for a Python ML project. We apply statistical methods and novel clustering techniques, including time series analysis and noise reduction—methods less commonly used in software engineering research—to identify common refactoring patterns across different scopes of the development process. Moreover, through an in-depth analysis of code features at various granularities, we provide a detailed examination of the side effects associated with the identified refactoring patterns. Leveraging machine learning techniques, feature engineering, and addressing challenges related to imbalanced datasets, we introduce MLRefScanner—a tool designed to detect refactoring-related commits in Python projects, specifically within the machine learning (ML) domain, providing the first comprehensive analysis of refactoring operations within ML technical domain. Specifically, we hypothesize that (1) there are extended refactoring practices used by developers in both long-term and short-term contexts, (2) the adoption of refactoring patterns can be influenced by software releases within the CI/CD pipeline, and (3)

refactoring practices can vary across different languages and domains. To investigate these hypotheses, this thesis conducts large-scale studies to (1) identify common refactoring rhythms and tactics and measure their relationship with code quality, (2) examine release-wise refactoring patterns and their impact on release quality, and (3) study refactoring practices in Python by developing an ML-based prototype tool for refactoring detection in Python machine learning projects. The findings of this thesis will help practitioners to (1) understand existing refactoring practices at different stages of development, (2) comprehend their relationship with code quality, and (3) facilitate future research into refactoring practices in languages beyond Java, exploring potential differences and extensions in these languages.

1.4 Thesis Objectives

This section describes the goals of this thesis in addressing the previously mentioned challenges. Figure 1.5 provides a high-level overview of the thesis, mapping the objectives outlined below to the forthcoming chapters.

Objective I: Examine refactoring rhythms and tactics and their relationship with code quality. Developers often follow specific patterns in their weekly routines, such as focusing on development during weekdays and reserving refactoring for weekends. By collecting a large-scale dataset of 196 open-source projects and applying clustering techniques and statistical tests to compare the frequencies and significance of refactoring activities that deviate from the development process on weekdays, this thesis explores common refactoring rhythms. Furthermore, by incorporating time series clustering techniques on refactoring frequencies at different stages

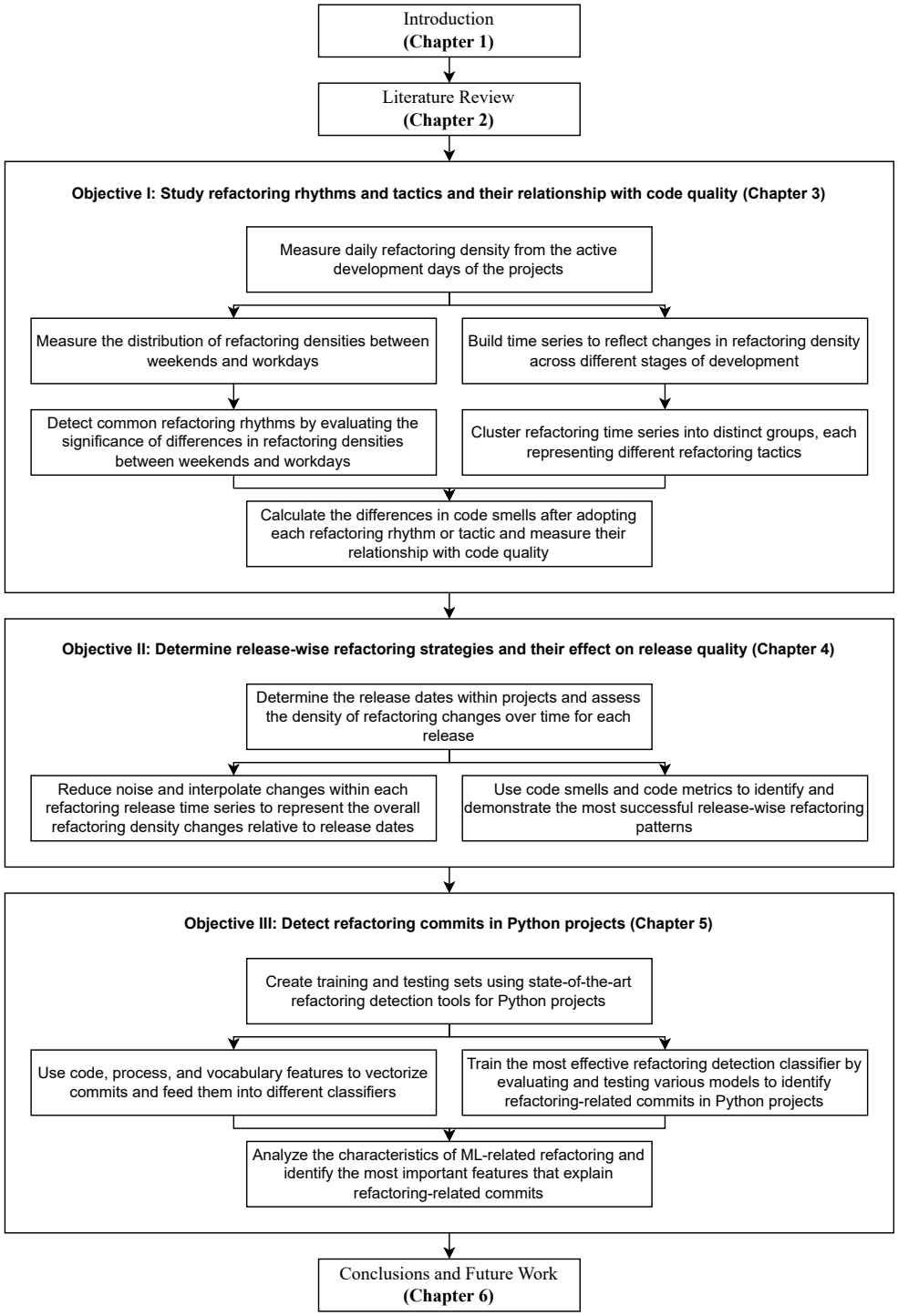


Figure 1.5: Thesis outline.

of the software development process (*e.g.,* early stages), this thesis complements existing refactoring tactics (*i.e.,* floss and root canal) and investigates extended refactoring tactics. Finally, the impact of the identified refactoring rhythms and tactics on various aspects of code quality is assessed, with a detailed examination of different types and dimensions of code smells. The overall objective is to provide insights into common refactoring rhythms and tactics and to offer actionable recommendations for their effective implementation.

Objective II: Identify release-wise refactoring strategies and their impact on release quality. The adoption of CI/CD practices accelerates release cycles compared to traditional methods, which may take years to publish new versions. This thesis conducts an in-depth analysis of release cycles by examining various release strategies (*e.g.,* release branch strategy) and their associated refactoring activities. By utilizing time series analysis and noise reduction techniques, it identifies common patterns that developers use to integrate refactoring tasks into software releases. These patterns are evaluated for their advantages and disadvantages with release quality, considering a comprehensive range of code smells and metrics. Finally, the thesis analyzes the transition from one refactoring-release pattern to another concerning its impact on code quality. The objective is to provide developers with actionable insights into the most beneficial refactoring practices within a CI/CD pipeline, aimed at enhancing release quality.

Objective III: Detect refactoring commits in Python projects. As Python has become one of the most popular programming languages, especially for machine learning libraries and frameworks, the absence of dedicated refactoring detection tools

limits the ability to analyze refactoring practices in Python projects. This thesis introduces an ML-based prototype tool that uses machine learning techniques to overcome the limitations of prior refactoring detection tools, which often require a complete abstract syntax tree (AST). This prototype tool is designed to detect refactoring-related commits in both machine learning and general Python projects. It utilizes a comprehensive set of features and advanced preprocessing techniques to identify refactoring operations, offering valuable insights into refactoring practices within the Python machine learning domain. Our objective is to explore differences in refactoring practices across various domains and programming languages and to support future research in less-studied languages, such as Python.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2 provides a literature review of related work to this thesis.

Chapter 3 presents our study on common refactoring rhythms and tactics and their relationship with code quality. This chapter details the data collection process, including refactoring operations, project features, and developer features. It outlines our preprocessing methods, such as the extraction of refactoring activities, measurement of refactoring density, and analysis of project and developer features. Finally, it describes our approach to addressing the research questions related to identifying refactoring rhythms and tactics and their relationship with code quality.

Chapter 4 outlines our approach to detecting refactoring patterns specific to each release within the CI/CD pipeline. This chapter details methods for collecting

refactoring and release data across various release strategies and demonstrates the steps for constructing refactoring time series. It presents our findings on release-specific refactoring patterns, highlights common refactoring practices within releases and their relationship with code quality, and identifies the most efficient transitions between these patterns. Finally, the chapter offers actionable recommendations for integrating refactoring practices into CI/CD pipelines.

Chapter 5 introduces our prototype tool for detecting refactoring-related commits within Python projects, specifically in the ML technical domain. This chapter details the process of constructing our training and testing datasets, including the extraction and preprocessing of process, code, and vocabulary features. It provides insights into the unique characteristics of refactoring within the machine learning domain, evaluates the model's performance, and compares our approach to state-of-the-art methods. Finally, it discusses the generalizability of our approach and its potential extension to other languages and technical domains.

Finally, **Chapter 6** concludes the thesis and outlines future directions for our work.

Chapter 2

Literature Review

In this chapter, we provide an overview of the prior studies related to the experiments conducted in this thesis. In particular, we introduce the related work along five directions: (i) refactoring strategies (Section 2.0.1), (ii) refactoring detection (Section 2.0.2), (iii) self-admitted refactoring (Sections 2.0.3), (iv) refactoring and code quality (Section 2.0.4), and (v) release and maintenance effort (Section 2.0.5).

2.0.1 Refactoring Strategies

In this section, we provide a background on previous studies related to refactoring strategies and their impact on software quality within the software development cycle.

Zhang *et al.* [56] conduct a survey study on developers for working overtime. The authors find that working overtime is a common behavior among software practitioners. Developers who work more often on weekends believe that working overtime could increase their productivity. Similarly, Claes *et al.* [57] study the frequency of the commit messages on 86 open-source projects and find that one-third of developers work overtime either at night or during the weekends. In terms of researchers, Wang *et al.* [58] study the download information of scientific papers and find that

many researchers work on weekends. However, the amount of overtime work differs among countries. Binnewies *et al.* [59] conducts a survey study on 133 employees and shows that psychological detachment, relaxation, and mastery experiences during the weekend are associated with being recovered for the upcoming week. Being recovered affects the weekly task performance, personal initiative, organizational citizenship behavior, and low perceived effort.

For long-term refactoring strategies, terms *floss* and *root canal* are two refactoring tactics identified in previous studies [5, 43]. *Floss* refactoring is distinguished by frequent refactoring, blended with the software development process, while the *root canal* is identified by occasional periods of refactoring which is not consistent with the software development process. Liu *et al.* [60] investigate refactoring histories on data collected from 753,367 engineers and suggest that between *floss* and *root canal*, the most frequently adopted refactoring tactic by engineers is *floss*. Sousa *et al.* [44] classify refactoring as *floss* and *root canal* and conduct a study on software projects to examine refactoring opportunities indicated by code smells. Murphy *et al.* [43] find that the tools are not aligned with floss tactics and are therefore not suitable for floss refactoring. It suggests that floss refactoring is likely to result in higher quality and lower costs in the long run.

Summary

Prior research has explored working rhythms and their impact on factors such as productivity. Additionally, it has introduced and applied the abstract refactoring concepts of floss and root canal tactics, relating them to code quality. In this thesis, we conduct a large-scale empirical study to first identify refactoring rhythms and variations within floss and root canal tactics, and then assess their relationship with code quality.

2.0.2 Refactoring Detection

A variety of tools have been proposed to detect refactoring operations in the history of software [19, 21, 61–63]. The main idea behind these approaches is to compare different versions of the code fragments stored in a version control system and point out refactoring operations. These tools can help us study refactoring activities on a large scale in the software maintenance process. Kim *et al.* [64] introduce Ref-Finder, which takes two versions of a program as input from workspace snapshots or subversion of a repository and extracts logical facts about the syntactic structure of a program. Nevertheless, Soares *et al.* [65] conducts a study and show that Ref-Finder has low precision and recall which leads to false-positive results, which means it is inaccurate in detecting refactorings. However, Tsantalis *et al.* [21] design a tool, Rminer, that overcomes the above constraints. Similarly to Ref-Finder [64], Rminer [10, 21] takes two revisions of source code from the commit history in the version control system of a Java project and returns a list of refactoring operations applied between two versions. Using a method similar to Rminer, Shiblu *et al.* [63] introduce jsDiffer, which is capable of identifying 18 types of refactoring for JavaScript. Sagar *et al.* [66] use text and code features to detect six types of refactoring operations in the history

of Java code. Alizadeh et al. [67] introduce a bot integrated into a version control system that monitors software repositories and identifies refactoring opportunities by analyzing recently changed files through pull requests. It then finds the best series of refactorings to fix the quality issues. Silva *et al.* [61] present RefDiff, a refactoring detection tool designed for Java, C, and JavaScript, capable of identifying up to 10 types of refactoring. Moghadam*et al.* [62] introduce RefDetect, which can detect up to 27 refactoring types in Java and C++. Atwi*et al.* [19] introduce PyRef, capable of identifying up to 11 refactoring operations in Python. Refactoring detection tools for Python are limited, and to the best of our knowledge, code conversion [68] and AST pattern matching [19] are the only approaches available. Dilhara *et al.* [68] convert Python code into Java and then use Rminer [10] to detect the refactoring operations in Python projects.

Summary

Various tools have been proposed for detecting refactoring in software history across different programming languages. These tools primarily rely on pattern-matching algorithms and rules to identify refactoring operations between different versions of source code. While refactoring detection tools for Java can identify a broader range of refactoring types, those available for other languages often suffer from limited coverage, failing to detect all types of refactorings in the development history. In this thesis, we propose an ML-based refactoring detection prototype tool for detecting refactoring in Python projects and compare our approach with state-of-the-art methods, demonstrating how our method addresses the limitations of existing tools in detecting refactorings.

2.0.3 Self-admitted Refactoring

AlOmar *et al.* [29] conduct an experimental study and identify a set of SAR patterns potentially used to describe refactoring commits. They find that commits with SAR patterns make a more significant contribution toward major refactoring activities. Ratzinger *et al.* [30] identify refactoring commits by investigating keywords like *"refactor"* and its extensions, such as *"needs refactor"* in the commit messages. Similar to Ratzinger *et al.* [30], Stroggylos and Spinellis [69] examine commit messages, searching for words stemming from *"refactor"* and label them as refactoring commits. Zhang *et al.* [31] distinguish the refactoring commits using 22 keywords adopted from Fowler [27] refactoring categories. AlOmar *et al.* [70] investigate self-affirmed refactoring commits by leveraging the keywords obtained from their previous work [29] to investigate self-affirmed refactoring commits in Java projects. Kim *et al.* [32] conduct a survey study, asking developers what keywords they use to indicate refactoring in their commits. They then utilize 10 keywords to detect refactoring commits from version history and correlate them with code features.

Summary

Previous studies have utilized a set of keywords to search through commit histories and identify refactoring-related commits. These studies either employed keywords directly or defined specific sets to reveal refactorings within commits. In this thesis, we use a machine learning technique that integrates keyword-based metrics with code and process metrics to detect refactoring activities within commit histories. We compare our approach with existing keyword-based methods for detecting refactoring commits. Our analysis demonstrates that keyword selection in commit messages alone is insufficient for detecting all possible refactoring commits.

2.0.4 Refactoring and Code Quality

Prior work has performed studies regarding the relationship between refactoring and code quality. Almogahed *et al.* [71] examine the studies that identify the impacts of code refactoring on software quality. It identifies that researchers agree that refactoring has a positive impact on both internal and external quality attributes. Moreover, Lacerda *et al.* [72] conducts a literature review on refactoring tools and common code smells to measure the relationship between refactoring operations and code smells. By analyzing the initial and final code smells after refactoring, the study finds that a significant proportion of code smells get eliminated after performing refactoring, which in turn preserves or enhances software quality during the maintenance process. Moreover, it notices that code smells and refactoring are linked by quality attributes and quality attributes that affect code smells are the same ones that affect refactoring. Bibiano *et al.* [13] correlate and study the effect of batch refactoring on code smells. It identifies that there is usually more than one refactoring operation

required to eliminate the code smells. Cinn'eide *et al.* [38] conduct a survey study on the benefits of refactoring and argue that, although refactoring is commonly believed to aim at removing code smells, developers are not strongly motivated by the desire to eliminate them. Murphy *et al.* [43] define two refactoring tactics, floss and root canal, using a dental metaphor. Floss involves frequent refactoring with other program changes, while root canal involves infrequent, longer periods of refactoring with few other program changes. Murphy *et al.* propose five principles and evaluate tools for alignment with floss tactics. Murphy *et al.* find that the tools are not aligned with floss tactics and are therefore not suitable for floss refactoring. It suggests that floss refactoring is likely to result in higher quality and lower costs in the long run. However, it does not propose a quantitative approach to measure this claim. Fontana *et al.* [73] explore refactoring tools and measure their limitations in removing code smells. Cinnéide *et al.* [38] report that even though refactoring is aimed at eliminating code smells, developers are not motivated to eliminate code smells by refactoring.

Summary

Prior studies have linked refactoring with the reduction of code smells and an increase in code quality. In this thesis, we explore the effectiveness of identified refactoring patterns with code quality and present large-scale studies on the relationship between refactoring patterns and code quality.

2.0.5 Release and Maintenance Effort

Previous studies have shown that the software maintenance process and refactoring can be closely related to the release of new versions. Baumgartner [74] utilizes a

Large Language Model (LLM) and develops an AI-driven pipeline to eliminate data clumps during releases. Saidani [75] investigates the impact of refactoring after adopting CI/CD and observes a decline in refactoring frequency and application following the implementation of CI/CD pipelines. Ding et al. [76] examine release testing and explain how tests may fail to accurately reflect the code's performance.

Summary

Prior studies have linked release cycles with refactoring and explored how refactoring can be influenced by CI/CD practices. However, they often lack empirical evidence or fail to provide explicit patterns based on large-scale data. In this thesis, we conduct a large-scale empirical study to identify common release-wise refactoring patterns and study their relationship with release quality.

Chapter 3

An Empirical Study on Refactoring Rhythms and Tactics

This chapter describes our study on common refactoring rhythms and tactics and their relationship with code quality. Section 3.1 presents the research problem and motivation of our study. Section 3.2 describes the setup of this study. Section 3.3 presents our approaches and results for answering our research questions. Section 3.4 discusses the threats to the validity of our findings. Finally, we summarize this chapter and outline future research directions in Section 3.5.

3.1 Problem and Motivation

Several factors contribute to the quantity of refactoring operations performed to improve code quality, such as developer perceptions, team experience, development schedule, software characteristics, and so forth [77]. Different teams may apply different refactoring strategies in short-term or long-term periods [16, 17]. Identifying patterns in refactoring practices and their relationship with code quality can help software developers adopt the most suitable patterns in their projects. More specifically

refactoring patterns have two perspectives: (1) refactoring rhythms and (2) refactoring tactics. Refactoring rhythms describe how refactoring operations split across the weekdays and usually focus on existing tasks. Refactoring tactics are referred to as long-term refactoring more focused on future development [78].

In the context of refactoring rhythms, existing studies focus on development rhythms and categorize development rhythms as work-day (*i.e.,* Monday to Friday) development and all-day development. Moreover, they correlate development rhythms with the measures, such as task performance and productivity [56,57,59]. However, to the best of our knowledge, no study has investigated the identification of refactoring rhythms and their relationship with code quality.

In terms of refactoring tactics, existing studies divide refactoring tactics into *floss* and *root canal* [5, 43, 44]. *Floss* refactoring is distinguished by frequent refactoring along with the development process. *Root canal* refactoring is identified by occasional refactoring aside from the development process. While the terms *floss* and *root canal* are widely used as refactoring tactics, the existence of other possible tactics and their relationship with code quality is not explored in the existing work.

In this chapter, we study developers' refactoring activities (rhythms and tactics) and their impact on code quality. To identify refactoring rhythms and other possible refactoring tactics, we study 196 Apache projects and introduce two metrics: daily refactoring density (DRD) and weekly refactoring density (WRD). Using the introduced metrics, we divide each project into daily and weekly time frames to identify frequent refactoring tactics and rhythms during the lifetime of a project. Then, we investigate the relationship between different rhythms and tactics with code quality. Such information can guide developers in selecting the most suitable and high-quality

refactoring rhythms or tactics for their projects. To this end, we aim to answer the following research questions:

***RQ*$_{3.1}$**: What are the rhythms of refactoring?

We analyze if the refactoring rhythms fit into the software development rhythms introduced by previous studies (*work-day* and *all-day*). By utilizing the DRD metric and performing statistical tests, we observe that the majority of projects (95%) apply two primary rhythms: (1) work-day refactoring (11%) and (2) all-day refactoring (84%).

***RQ*$_{3.2}$**: What are the most frequent refactoring tactics used in projects?

To identify refactoring tactics, we utilize the WRD metric and form a time series of refactoring activities for every project. Using time series clustering techniques, we cluster refactoring time series and we observe four variations of floss and root canal refactoring tactics:

- **Intermittent spiked floss:** Regular and consistent refactoring with fewer sudden increases (spikes) compared to frequent spiked floss.
- **Frequent spiked floss:** Consistent refactoring but with more spikes in refactoring density compared to intermittent spiked floss.
- **Intermittent root canal:** Once in a while refactoring in high densities, but with most weeks having no refactoring densities.
- **Frequent root canal:** More frequent refactoring with more spikes in refactoring density compared to intermittent root canal with most weeks having no refactoring activities.

***RQ*$_{3.3}$**: What is the relationship of different refactoring rhythms and tactics with code quality?

> In the two first research questions of this chapter, we identify frequently used refactoring rhythms and tactics. Furthermore, we are interested in understanding how different rhythms and tactics are associated with code quality improvement (*i.e.,* reducing code smells). To examine the relationship between refactoring rhythms with code quality metrics, we use statistical tests to rank and cluster code smell changes after adopting each refactoring rhythm and tactic. We observe that root canal-based tactics are more targeted refactoring operations and, therefore, are correlated with more reduction of code smells compared to floss-based tactics and deliver higher quality code. Furthermore, we observe that refactoring rhythms are not significantly correlated with software quality. Consequently, refactoring rhythms are chosen based on the project assets and the development team's comforts. Finally, we provide some guidelines on positive and negative relationships between different refactoring tactics and different types of code smells.

The replication package for this chapter is available online[1].

3.2 Experiment Setup

This section presents the setup of our study, including our data collection and data analysis approaches.

[1] https://github.com/Shayan-Noei-PhD-Replication-Packages/Chapter-3-Strategies

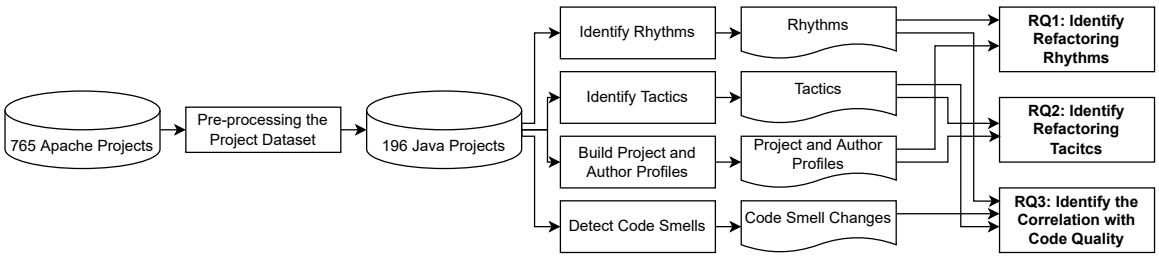


Figure 3.1: Overview of our empirical study on refactoring rhythms and tactics.

3.2.1 Overview of Our Approach

Figure 3.1 gives an overview of our study. We conduct our research using the projects with a reasonable amount of development activities from the 20-MAD Apache dataset [79]. We extract the refactoring history of these projects and calculate refactoring density metrics along with their lifespan. By utilizing refactoring density metrics, we compare refactoring distributions at different time periods to answer the first and second research questions that aim to identify refactoring rhythms and tactics. Furthermore, we use the characteristics of the projects and developers to provide insights into the rationale for using such rhythms and tactics. Finally, by measuring quality changes after adopting each rhythm and tactic, we identify the relationship of the identified rhythms and tactics with the quality of code.

3.2.2 Data Collection

To perform our experiments, we perform several steps to collect our dataset.

Project Selection and Pre-processing

The 20-MAD Apache dataset [79] contains information about commits and issues related to 765 Mozilla and Apache projects with a timespan of 20 years. In particular, the dataset contains 3.4M commits, 2.3M issues, and 17.3M issue comments. Considering that Java is one the most popular programming languages [24, 80] and it is best supported by refactoring extraction tools (*e.g.,* Rminer [10]), we limit our study to Java projects. To select projects with enough data that help identify the different refactoring tactics and rhythms, we exclude the projects that:

- have less than 80% of Java source code;
- have less than the 1^{st} quantile of commit counts (*i.e.,* $<$ 1,021 commits); and
- have a short lifespan (*i.e.,* $<$ one-year of commit history).

As a result, we obtain 196 Java projects that have sufficient commit history and lifespan for our analysis.

The studied projects in our dataset have varying lifespans and therefore, possess different development histories. Furthermore, refactoring habits or requirements may change over time. For example, as a project ages, design issues may be fixed less frequently [81]. Hence, we cannot compare refactoring practices unless we have projects with a similar lifetime. To be able to analyze the refactoring activities in the projects, we partition longer projects into multiple stages. We utilized the first, second, and third quartiles of project ages to establish thresholds and divided all projects into four age groups, with each group containing an equal number of projects. The age groups are categorized as follows: (1) younger than 4.5 years (*i.e.,* the first quartile), (2) between 4.5 (first quartile) and 7 years (second quartile), (3) between 7 (second

quartile) and 8.5 years (third quartile), and (4) older than 8.5 years (third quartile) of activities. We excluded projects with less than 4.5 years of activities from our study because such projects have a wide variety of ages, making it difficult to compare similar activities among them, unless they have a similar lifespan. By using the identified age groups, we define different stages of software development: (1) early stage: start of a project until the 4.5^{th} year, (2) middle stage: 4.5^{th} year of a project until the 7^{th} year, and (3) late stage: 7^{th} year of a project until the 8.5^{th} year which is the longest age considered in all studied projects. As a result, we observe that 50 projects have ages between 4.5 and 7 years, 49 projects have ages between 7 to 8.5 years, and 50 projects have ages of more than 8.5 years. To compare the old projects (*e.g.,* 10 years old) with young ones (*e.g.,* 5 years old), we divide the older projects into two or three stages using the thresholds of the age groups. For instance, if a project has 6 years of activities, it has only the early stage of development (*e.g.,* the first 4.5 years of activities), while a project with 10 years of activities has the early, middle, and late stages of development. Doing so allows us to identify similar rhythms and tactics that might appear among the projects at different stages. We use these three development stages throughout the analyses performed in this study.

Refactoring Extraction

To extract the refactoring operations, we use the Rminer 2.0.3 tool [10], which is an AST-based algorithm that finds up to 59 Java refactoring types from the commit history without the need for user-defined thresholds [10, 21]. Furthermore, Rminer is a superior refactoring detection tool compared to its opponents and identifies refactoring operations with an overall precision of 99.7% and a recall of 94.2% [10]. To

measure the accuracy of the tool on our dataset, with a confidence level of 95% and a margin of error of 5% on the total number of commits, we select 385 commits to perform our manual validation. In each commit, we reviewed all types of refactorings that the tool could identify and determined if and how many of them were present in the results of the tool. We then compared our results with the tool's results. For example, if we identified a pull-up method but the tool did not, we marked it as a tool failure and vice versa. The manual validation is done by the author of the thesis and one undergraduate computer science student. The results of the manual validation show an overall precision of 97%, recall of 96%, and F1 score of 95% respectively. Furthermore, we calculate Cohen's kappa coefficient [82] from the participant's manual validation results and achieve a score of 0.91, which suggests a strong agreement. Therefore, we run Rminer on every selected project for each commit to extract refactoring activities in the history of development.

Code Smells Extraction

To identify the relationship between the refactoring rhythms or tactics and code quality, we need to analyze how these refactoring patterns reduce or increase the frequency of code smells. As we use code smells as code quality indicators, we analyze how refactoring rhythms and tactics are associated with the reduction or increase in the frequency of code smells.

We use the Designite tool [83] to extract code smells. Designite can identify the most types of code smells compared to its alternatives and detects numerous code smells in large codebases [83, 84]. For instance, Arcan [85] and Hotspot detector [86] can detect only 4 types of code smells. Similarly, Jdeodorant [87] detects 5 and

Arcade [88] detects 11 types of code smells. Designite can identify 35 code smell types including 7 architecture smells, 18 design smells, and 10 implementation smells as listed in Table 3.1. The different categories of code smells have specific causes and impacts on the software system. Architecture code smells arise from poor software designs within the system architecture and can negatively impact system quality, performance, and lifespan [89]. Design code smells result from inadequate system design and have a negative impact on code quality [35]. Implementation code smells, on the other hand, stem from poor implementation decisions made by contributors and can negatively affect the quality of the code [5]. To address all three types of code smells, refactoring has been shown to be an effective solution [5, 35, 89, 90]. Being able to identify the various types of code smells allows us to study the impact of different refactoring strategies and techniques in a more comprehensive manner. Moreover, Designite is open-source and can be used on cloned projects at the code level. Therefore, we use Designite to measure code smells at the start and end of each development stage. This helps us evaluate the frequency and types of code smells before and after implementing each refactoring tactic or rhythm. Given that development activities are minimal in the early days of a project, we use the first quartile of the early stage as the starting point and consider it the initial quality point for extracting code smells.

3.2.3 Author and Project Profiles Identification

Different teams may have different author formations with distinct skills. Furthermore, the characteristics of the projects, such as their size, may lead to the adoption

Table 3.1: The list of code smell metrics used in the study.

Category	**Metrics**
Architecture Smells	Cyclic Dependency, Unstable Dependency, Ambiguous Interface, God Component, Feature Concentration, Scattered Functionality, and Dense Structure
Design Smells	Imperative Abstraction, Wide Hierarchy, Broken-Modularization, Cyclic Hierarchy, Hub like Modularization, Multipath Hierarchy, Unnecessary Abstraction, Missing Hierarchy, Multifaceted Abstraction, Feature-Envy, Unutilized Abstraction, Rebellious Hierarchy, Deficient Encapsulation, Broken Hierarchy, Unexploited Encapsulation, Insufficient- Modularization, Cyclically Dependent Modularization, and Deep Hierarchy.
Implementation Smells	Long Method, Complex Method, Long Parameter List, Long Identifier, Long Statement, Complex Conditional, Abstract Function Call from Constructor, Empty Catch Clause, Magic Number, and Missing Default.

Table 3.2: The list of author metrics used in the study and their descriptions.

Author Metrics	**Description**
Contribution	Defines the code churn of the developer.
Timezone	Describes the primary timezone of a developer.
Experience	Describes the time that a developer contributes to a project.
Commits	Defines the number of commits submitted by a developer.
Work Time	Explains the primary time of commit submission (*e.g.,* 14:00).
RefactoringDensity	Describes the density of contribution toward refactoring.

of different refactoring strategies. Therefore, it is essential to identify the types of authors and projects in different stages of development to gain insights into the adopted refactoring tactics and rhythms. To this end, we pick a set of metrics to explain the characteristics of the authors and projects. Furthermore, we cluster authors and projects based on the collected metrics and form different profiles for projects and authors. The selected metrics are listed in Tables 3.2 and 3.3.

We collect contribution, timezone, experience, commits, work time, contributors count, and the age of the projects by writing a python script and traversing the

Table 3.3: The list of project metrics used in the study and their descriptions.

Project Metrics	**Description**
Files	Describes the total number of files.
Comments	Defines the lines of comments added to the codebase.
Lines of Code	Describes all lines of codes written in Java.
Contributors	Describes the total number of known contributors.
Timezones	Defines the number of different timezones of the developers contributing to the project.
Commits	Describes the total number of commits.
Age	Defines the length between the first commit and the last commit in days.
Stars	Describes the popularity of a project in terms of stars gained.
Refactoring Density	Describes the density of refactoring in a repository.

commit logs. To calculate the refactoring density, we used Rminer to obtain the number of refactoring lines and a bash script to calculate the total code churn (*i.e.,* all lines of code added or deleted in a commit). We then divided the number of refactoring lines by the total code churn. Moreover, we obtain information about files, comments, and lines of code from the Cloc tool [91] and we use GitHub API [92] to fetch stars.

Highly correlated metrics are linearly related and can be expressed by each other. Furthermore, redundant metrics can be derived from other metrics. Having highly correlated or redundant metrics makes it difficult to analyze the impact of the metrics [93]. Thus, we perform correlation analysis and redundancy analysis to remove correlated and redundant metrics.

- **Correlation analysis:** We find that the author and project profile metrics do not follow a normal distribution, thus we use Spearman's rank correlation coefficient to find the correlation between the computed metrics. A coefficient of > 0.7 represents a high correlation [94]. For each pair of highly correlated

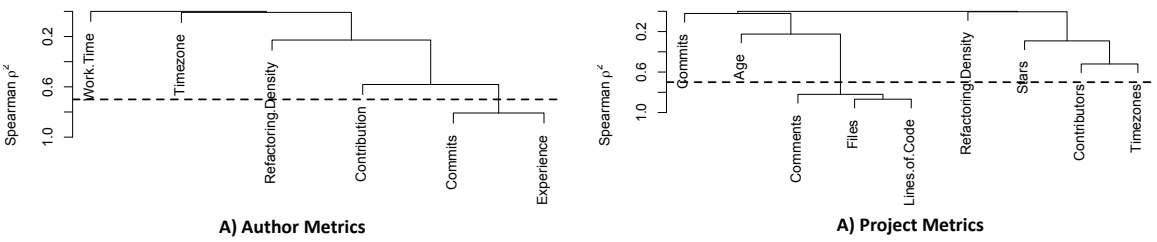


Figure 3.2: The results of the correlation analysis of author and project profile metrics.

metrics, we remove one metric and keep the other in our model. Figure 3.2 shows a dendrogram representing the correlation between the project and author metrics.

- **Redundancy analysis:** R-square is a measure that shows how variance of a dependent variable can be explained by independent variables [95]. We use an R-squared cut-off of 0.9 to identify redundant metrics that can be estimated from other metrics.

We measure highly correlated and redundant metrics in the project and developer profiles metrics and exclude them from the studied metrics. The correlation (Figure 3.2-A) analysis reveals that author's *commits* and *experience* are highly correlated. Likewise, in project metrics, *files*, *comments*, and *lines of codes* are highly correlated (Figure 3.2-B). Hence, we remove project's *comments*, project's *lines of codes*, and author's *experience*. After removing highly correlated metrics from the clustering set, we performed redundancy analysis, but no redundant metrics were identified.

After removing the highly correlated and redundant metrics, based on the distribution of each metric in different quartiles, we divide them into four groups and label them as *Least*, *Less*, *More*, and *Most* [96]. We use k-mods clustering, an extension of

Table 3.4: The clusters identified by K-mods clustering to identify author profiles.

Label	**Timezone**	**Contribution**	**Refactoring Density**	**Commits**	**Work Time**
Main Authors	Less (-5.00, 0.00]	More (577.00, 9,848.0]	Most (0.19, 1.00]	More (7.0, 63.0]	More (14:57, 17:06)
Casual Contributors	Least [-12.00, -5.00]	Least [0.00, 23.00]	Least [0.00, 0.00]	Least [0, 2]	Most (17:06, 24:00]
Core Authors	Less (-5.00, 0.00]	Most (9,848.0, ∞)	More (0.06, 0.19]	Most (63.0, ∞)	Less (12:33, 14:57]

the k-means [97] clustering algorithm, which is suitable for clustering categorical data, to cluster the labeled metrics and cluster them into different profiles. We use elbow method [98] to find the optimal number of clusters (k) and manually validate and check if the clustering results provide distinct centroids. The elbow method involves plotting a graph that displays the number of clusters versus the sum of squared errors (SSE) for each cluster. The optimal number of clusters is identified by the point on the graph where the SSE begins to level off and form an elbow shape [99].

For Author profiles, we identify 3 as the optimal number of clusters with the optimal cost value of 62,872. Therefore, we apply k-mods with 3 clusters (k=3) and identify three major profiles. Based on the selected metrics (*i.e.*, timezone, contribution, refactoring density, commits, and work time), we label the three identified clusters as *main authors*, *casual contributors*, and *core authors*. The *core authors* make the most contributions while the *casual contributors* make the least contributions. The *main* and *core* authors are primarily located in America and Western Europe with commits between 12:33 to 17:06 at their local time (14:57-17:06 for *main authors* and 12:33-14.57 for *core authors*). Casual contributors are primarily located in North America and commit from 17:06 to 24:00 (midnight).

Table 3.5: The clusters identified by K-mods clustering to identify project profiles.

Label	**Files**	**Contributors**	**Timezones**	**Commits**	**Age**	**Stars**	**Refactoring Density**
Vibrant	Most (2157.75, ∞)	Most (44.00, ∞)	Most (10.75, ∞)	Most (3,450.75, ∞)	Less (902.77, 1,684.16]	Most (237.25, ∞)	More (0.15, 0.20]
Maintaining	Least [0.00, 530.75]	Least [0.00, 15.00]	Least [0.00, 1.00]	Least [0.00, 951.00]	Less (902.77, 1,684.16]	Less (1.00, 38.00]	Most (0.20, 1.00]
Obsolete	Less (530.75, 1,242.00]	More [27.00, 44.00]	More [5.00, 10.75]	Less (951.00, 1,702.00]	Most (2,096.51, ∞)	More (38.00, 237.25]	Least [0.00, 0.10]
Growing	More [1242.00, 2157.75]]	Least (15.00, 27.00]	Least [0.00, 1.00]	More (1,702.00, 3,450.75]	More (1,684.16, 2,096.51]	Least [0.00, 1.00]	Least (0.00, 0.10]

For the Project profiles, we find four clusters that provide different meaningful clusters with the optimal cost value of 919,000. Hence, we apply k-mods clustering with k=4 and identify four major profiles, namely, *vibrant*, *maintaining*, *obsolete*, and *growing* projects. *Vibrant* projects have the most commits, contributors, and stars while *obsolete* projects experience the least refactoring density along with less commits, most age, and more stars. Furthermore, *growing* projects experience least refactorings with the least stars while having more commits and least contributors. Moreover, *maintaining* projects share least commits and contributors with the most refactoring density. Tables 3.4 and 3.5 summarize the results of author and project clustering.

3.2.4 Research Methods

This section presents the research methods applied to answer the research questions.

Refactoring rhythms identification

To identify the refactoring rhythms of the projects, we require a measure to describe the amount of refactoring activities (*i.e.,* refactoring churn) applied on each day of the week. As the refactoring activities could be reflected by the amount of code changes caused by refactoring, we use the refactoring churn to quantify the amount of refactoring activities and normalize it by the actual code churn. Therefore, we introduce daily refactoring density (DRD), which indicates the amount of refactoring activities deviated from the overall development (*e.g.,* the total code churn) of each day of development. The DRD is calculated as below:

$$\text{DRD}(i) = \frac{\text{Refactoring churn of the day } (i)}{\text{Total code churn of the day } (i)} \tag{3.1}$$

We measure the daily refactoring densities (*i.e.,* DRD) and compare them throughout the week. We form seven groups, each of which represents one day of the week and contains all refactoring activities that occurred on that particular day. Doing so allows us to find the similarities and differences of refactoring activities from one day to another. As our data does not follow a normal distribution, to measure the significance of the differences or similarities of the measured DRDs among different days, we use the Kruskal-Wallis test, an extension of Mann–Whitney U test [100] that evaluates if two or more samples come within the same distribution [101]. The Kruskall-Wallis test does not assume that the data is normally distributed or not. We use p-value > 0.05 to decide if a test of null-hypothesis is significant [102]. The null-hypothesis is a statistical theory that measures if a significant relationship exists between two sets of data [103].

Refactoring tactics identification

Previous studies show that the majority of projects utilize agile methodologies, which require small tasks to be finished within one week of development [102,104]. To understand the long-term refactoring activities applied by developers in the long run, we need a measure to understand the amount of refactoring churns compared to the development per week of the development. Hence, we propose a weekly refactoring density WRD metric, which reflects the amount of refactoring activities per week of development. The WRD metric is computed using the following formula:

$$\text{WRD}(i) = \frac{\text{Refactoring churn of the week } (i)}{\text{Code churn of the week } (i)} \tag{3.2}$$

For each project, we create a time series of WRDs within every stage of development. Each data point of the time series includes a week of development and the corresponding WRD metric for that week. Accordingly, the refactoring time series data depicts refactoring behaviors over time. Therefore, similar refactoring time series between different stages of the projects represent a similar refactoring habit. The created refactoring time series share different lengths and they vary in speed. For instance, the development period of Project A is 10 weeks, while that of Project B is 20 weeks, indicating a variation in their length. Additionally, Project A experiences a refactoring spike in the second week of development, while Project B experiences the same spike in the fifth week of development, indicating a variation in the speed of refactoring activities. Therefore, comparing the refactoring time series with point-to-point measures, such as euclidean distance [105], could not overcome the limitations of speed and length variation and could not identify similarities optimally. Therefore, we use the dynamic time warping (DTW) algorithm to measure the similarity between the refactoring time series of project stages as refactoring tactics. DTW overcomes the limitation of point-to-point comparisons with the step pattern that allows transitions and weights between two pairs [106]. Moreover, DTW is an algorithm for calculating the similarity between two time series that vary in speed [107].

Quality Changes Measurement

For identifying the relationship between the refactoring rhythms or tactics and code quality, we need to analyze how these refactoring rhythms or tactics reduce or

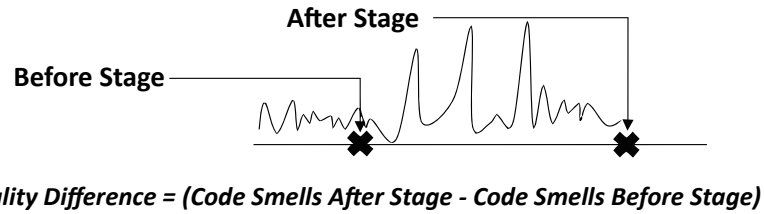


Figure 3.3: How changes in code smells are calculated after each stage of development in each project.

increase the frequency of each type of code smell. To do this, we use the boundary points before and after each stage of development. As it is shown in Figure 3.3, we measure the frequency of each code smell type before and after each stage (*i.e.,* between two consecutive stage boundaries) and measure the difference (reduction or increase) in each type of code smell.

Since a larger codebase could contain more code smells, we normalize the frequency of code smells by the lines of code (LOC) in the codebase. Additionally, since more code changes (*i.e.,* larger code churn) may lead to larger differences in code smells, we normalize the difference in code smells by the code churn within each stage. Finally, we calculate the differences in each type of code smell and label it according to the identified rhythm and tactic adopted in that stage. We utilize the frequency of code smells at the end of each stage (ECS), the lines of code at the end of each stage (ELC), the initial frequency of code smells in each stage (ICS), the lines of code at the beginning of each stage (ILC), and the total code churn of each stage (CC) to measure the normalized differences of the total frequency of code smells of each stage (CSD). To measure CSD at each stage (i), we use the following equation:

$$\mathrm{CSD}(i) = \frac{(\mathrm{ECS}(i)/\mathrm{ELC}(i) - \mathrm{ICS}(i)/\mathrm{ILC}(i))}{\mathrm{CC}(i)/\mathrm{ELC}(i)} \tag{3.3}$$

An increase in CSD indicates an increase in the number of code smells (*i.e.,* decreased code quality) and a decrease in CSD indicates an increase in the number of code smells (*i.e.,* increased code quality). To measure the smell difference after utilizing each refactoring rhythm or tactic, we require a multiple comparison method to cluster and rank the identified rhythm or tactic into statistically significant groups and rank them based on CSD metric (*i.e.,* changes in code quality). Therefore, we use the Scott-Knott-ESD [108, 109] test that uses a multiple comparison method that divides and ranks a set of input distributions into statistically distinct groups [109]. Scott-Knott-ESD is an extension of the Scott-Knott [110] test with the addition of effect size difference. The effect size examines the strength of the difference between different groups of data [111]. Therefore, we cluster and rank the refactoring rhythms and tactics based on the CSDs and identify the rhythms or tactics leading to a codebase with an increased or decreased amount of code smells.

3.3 Results

In this section, we provide the motivation, approach, and findings for each of our research questions.

3.3.1 $RQ_{3.1}$: What are the rhythms of refactoring?

Motivation

In software development, developers can have various working rhythms. For example, some developers prefer to work only on workdays; however, others do not mind working even on weekends. Existing studies focus on development rhythms and categorize development rhythms as *work-day* (*i.e.,* Monday to Friday) development

and *all-day* (*i.e.,* Monday to Sunday) development [56, 57, 59]. Prior research reports that the state of getting recovered during the weekend from working on the workdays is correlated with an increase in weekly task performance and personal initiative [59]. Moreover, the state of working overtime is associated with a decrease in productivity [112]. Furthermore, previous studies have suggested that deviating from regular development to perform refactoring may help address unhealthy code and potentially improve code quality [43]. Inspired by prior work, our intuition is that providing dedicated time for refactoring outside of regular development cycles enables developers to focus more on addressing unhealthy code through refactoring, resulting in improved code quality. Thus, we study the refactoring rhythms based on their deviations from the development rhythms. Understanding the refactoring deviations from the regular development rhythms and their relationship with the code quality improvement can assist software teams and developers to (1) understand the existing refactoring rhythms and (2) adopt/apply the most effective refactoring rhythms. In this research question, we investigate and characterize different refactoring rhythms to help developers understand the existing refactoring rhythms and identify which one suits their needs.

Approach

As described in Section 3.2.4, to identify the refactoring rhythms of the studied projects, we form seven groups and measure DRDs on every day of development. Moreover, we compare DRDs throughout the week to discover refactoring rhythms using the Kruskal-Wallis test [101].

Prior studies [56, 57, 59] divide the software development process into two groups—

work-day development and *all-day* development (including workdays and weekends). To identify the refactoring rhythms adopted in different stages and different projects, we need to compare different days of refactoring. Hence, we group DRDs into seven groups based on the weekdays, where each group represents a weekday and depicts the overall DRD distribution on the corresponding day of the week. Using the Kruskal-Wallis test, we first identify whether we can fit the majority of the rhythms adopted from the selected project stages into two groups, namely *work-day* refactoring and *all-day* refactoring. To do this, we perform two individual tests using the following hypothesis:

- $\boldsymbol{H_{0\text{-}1}}$**:** *Refactoring densities are similar among all days of the week.*
- $\boldsymbol{H_{0\text{-}2}}$**:** *Refactoring densities are similar among all workdays of the week*

To this end, after running the first test, we exclude the stages of projects that have a similar distribution of refactoring on all days of the week and perform the second test. We accept a hypothesis if the p-value is higher than 0.05 and reject otherwise.

Clustering project and developer profiles in terms of refactoring activities. To understand the distribution and significance of the refactoring rhythms across different project and author profiles, we rank the combinations of the different refactoring rhythms and project or author profiles. Using the Scott-Knott-ESD [108, 109], we group combinations of refactoring rhythms and project or author profiles into statistically significant clusters. Specifically, we perform two separate Scott-Knott-ESD clustering analyses: (1) for combinations of project profiles and refactoring rhythms, and (2) for combinations of author profiles and refactoring rhythms.

In the clustering method, each clustered item (*i.e.,* a node) represents the distribution of projects or authors that are associated with a specific refactoring rhythm.

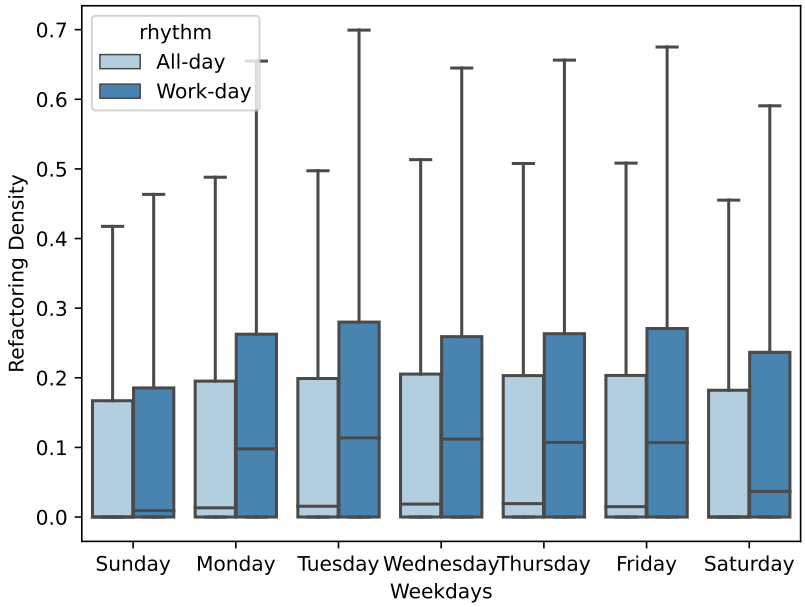


Figure 3.4: Comparison of refactoring density between *work-day* and *all-day* refactoring rhythms.

Moreover, each clustered item is represented by a vector of the same length as the number of projects or authors, with each project or author being assigned a value of 1 if it is associated with the specific project or developer profile and the specific refactoring rhythm, and a value of 0 otherwise. For example, All-day-Vibrant refers to the distribution of the *all-day* refactoring rhythm across all projects that fall under the *vibrant* project profile. Each cluster corresponds to a statistically significant distribution of the various combinations of refactoring rhythms and project or author profiles across the dataset. The results of the Scott-Knott-ESD test provide insights into how refactoring rhythms are distributed across different profiles.

Identifying refactoring rhythms characteristics. To study the differences between refactoring operations (*e.g.,* pull up method) performed on weekends and those

performed on workdays, we use the Mann–Whitney U test [100] to compare the distribution of each refactoring operation on the weekend and workdays. We utilize Cliff's Delta to measure the effect size of the differences. We consider the operations that obtain a p-value < 0.05 and an effect size > 0.33 [113], indicating a medium or large magnitude of difference, as the operations that are performed significantly differently between the weekends and the workdays.

Findings

The majority (95%) of project stages follow one of the work-day or all-day refactoring rhythms. We accept the null hypothesis $H_{0\text{-}1}$ for 84% of the project stages and the null hypothesis $H_{0\text{-}2}$ for 11% of the remaining project stages. For the 5% of the project stages, both null hypotheses $H_{0\text{-}1}$ and $H_{0\text{-}2}$ are rejected. Therefore, our analysis shows that only a few project stages (*i.e.,* 5%) do not follow any of the initial refactoring rhythms. Specifically, we find that 11% of the project stages perform *all-day* refactoring, whereas 84% of the project stages perform the *work-day* refactoring rhythm.

In the *work-day* refactoring rhythm, we observe a significant difference in refactoring densities between workdays and weekends, with the median refactoring densities being higher in workdays compared to weekends, as illustrated in Figure 3.4. Additionally, certain types of refactoring are applied differently between weekends and workdays, as shown in Figure 3.5-A. These types of refactoring include *move class*, *pull up method*, *pull up attribute*, *add attribute annotation*, *extract interface*, *add parameter annotation*, *modify parameter annotation*, *split attribute*, *move and rename attribute*, and *split parameter*. These refactoring actions mainly relate to the class/method

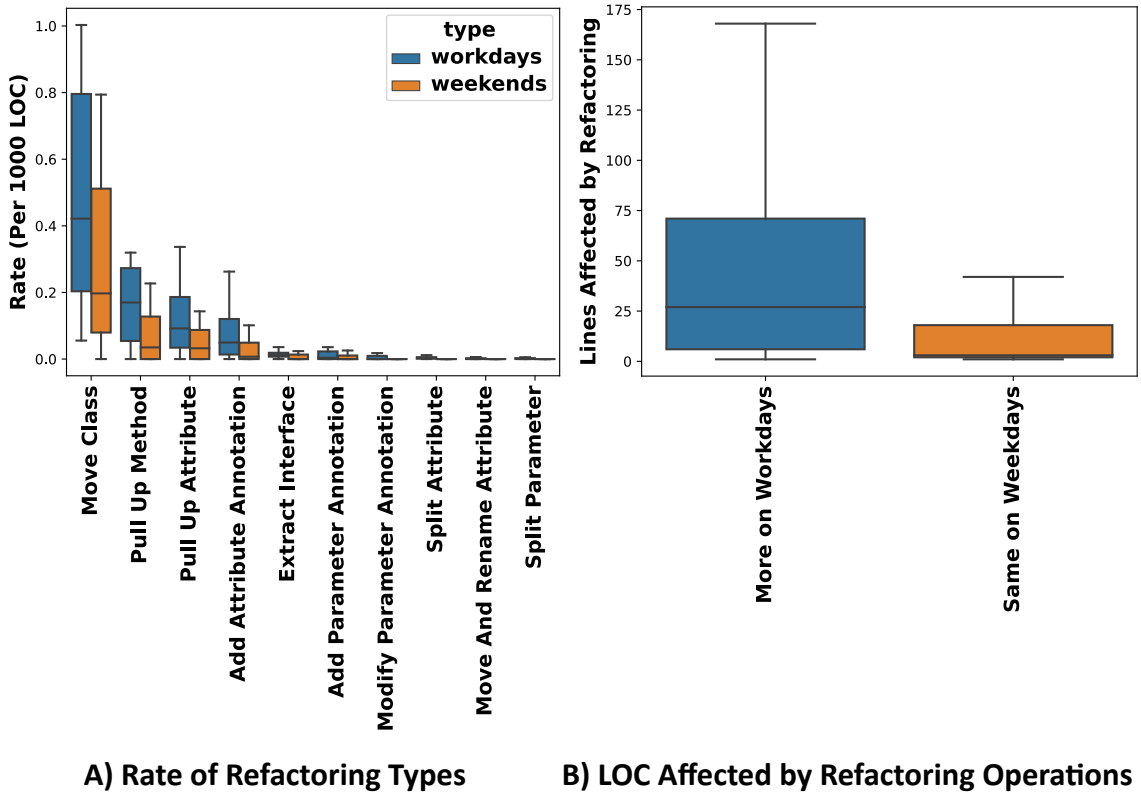


Figure 3.5: The different refactoring operations and the lines of refactored code applied in weekend compared to weekdays in *all-day* refactoring rhythm.

level and play a crucial role in shaping the overall system design. In contrast, other refactoring types, such as *extract method*, are applied consistently throughout the week. Therefore, developers may prefer to perform complex design-level refactorings on workdays, leaving weekends for less risky modifications. We observe that the median lines of the codes affected by the workday refactoring types (*i.e.,* the refactoring types that are applied more on workdays) are higher compared to weekday refactoring types (*i.e.,* the refactoring types that are applied similarly during workdays and weekends) (Figure 3.5-B). Therefore, developers apply more heavy-weight refactoring operations that involve more code changes (*e.g., move class*) during the

Table 3.6: Scott-Knott-ESD test results on the refactoring rhythms and associated project and author profiles.

Project Profiles			**Author Profiles**		
Rhythm Profile	**Cluster Rank**	**Mean (%)**	**Rhythm Profile**	**Cluster Rank**	**Mean (%)**
All-day Maintaining	1	0.89	All-day Core	1	0.79
All-day Growing	1	0.83	All-day Main	1	0.80
All-day Vibrant	1	0.83	All-day Casual	1	0.76
All-day Obsolete	1	0.82	Work-day Casual	2	0.19
Work-day Vibrant	2	0.14	Work-day Main	2	0.16
Work-day Obsolete	2	0.12	Work-day Core	2	0.15
Work-day Growing	2	0.11			
Work-day Maintaining	3	0.06			

workdays compared to the weekends.s In contrast, in the *all-day* refactoring rhythm, the Kruskal-Wallis test results demonstrate that there is no statistically significant difference in the density of refactoring among the different days of the week, and the median refactoring densities are consistent across all days of the week, as depicted in Figure 3.4. Additionally, we do not observe a significant difference in the refactoring operations performed on weekends compared to workdays. Therefore, it appears that developers exert an equal amount of effort towards refactoring throughout the week in the *all-day* rhythm.

Table 3.6 shows the results of the Scott-Knott test, providing further insights into the relationship between the project and author profiles with their corresponding rhythms:

Among project profiles: In the *maintaining*, *obsolete*, *growing*, and *vibrant* project stages, the *all-day* refactoring rhythm (Cluster 1) is often used with a similar distribution than the *work-day* refactoring rhythm with over 82% utilization practice (Clusters 2 and 3). In *vibrant*, *obsolete*, and *growing* project stages *work-day* refactoring rhythm (Cluster 2) is more frequently used compared to the *maintaining* project stages. *Maintaining* project stages experience the most refactoring density and the least number of commits (Table 3.3). Moreover, in *Maintaining* project stages, the usage of the *all-day* refactoring rhythm is highest, at 89% (Cluster 1), compared to the *work-day* refactoring rhythm, which is 0.06% (Cluster 3). Therefore, in *maintaining* project stages where there is less development and more refactoring, developers are likely to perform the *work-day* rhythm less frequently and focus on refactoring whenever they have time. The observation that the distribution of the *work-day* refactoring rhythm is similar in *vibrant*, *obsolete*, and *growing* projects (Cluster 2), and the distribution of the *all-day* rhythm is also similar in these project stages (Cluster 1), indicates that the project profile does not have a significant impact on the choice of refactoring rhythm in *vibrant*, *obsolete*, and *growing* project stages.

Among author profiles: *Core*, *main*, and *casual* authors often utilize the *all-day* refactoring rhythm with a similar distribution (Cluster 1). Moreover, the distribution of the *work-day* rhythm is similar in all author profiles (*i.e., core*, *main*, and *casual*) (Cluster 2). Since the distribution of different author profiles in *all-day* and *work-day* rhythms separately are similar, the choice of specific refactoring rhythm is not influenced by the type of developer. Therefore, it is likely that authors choose different rhythms based on their preferences.

$RQ_{3.1}$: Summary

Software projects follow two major refactoring rhythms: *work-day* and *all-day* refactoring. The *work-day* refactoring rhythm tends to have higher densities of refactoring to the code base from Monday to Friday. In the *all-day* refactoring rhythm, there is no significant difference in refactoring activities on different days of the week. In *maintaining* project stages, the *all-day* refactoring rhythm is more prevalent compared to other project stages. The choice of refactoring rhythms (*all-day* or *work-day*) is not influenced by the type of authors.

3.3.2 $RQ_{3.2}$: What are the most frequent refactoring tactics used in projects?

Motivation

Previous studies have only classified refactoring tactics as either *floss* or *root canal* [5, 43, 44]. *Floss* refactoring is distinguished by frequent refactoring along with the development process. On the other hand, *root canal* refactoring is identified by occasional refactoring aside from the development process. While the terms *floss* and *root canal* tactics have been useful in understanding the general patterns of refactoring, there may be other potential refactoring tactics that have not yet been identified. Moreover, understanding the distinctive features of each refactoring tactic can offer valuable insights into developers' decision-making processes when choosing a particular tactic. Additionally, recognizing various refactoring tactics can establish a common vocabulary for describing them, facilitating communication among practitioners. This, in turn, helps developers comprehend the refactoring tactics they use

and choose or switch to the most appropriate tactic for their project. In this research question, by considering different stages of development in different projects, we investigate whether there are more refactoring tactics other than *floss* and *root canal*.

Approach

As described in Section 3.2.4, to understand the refactoring tactics in the studied projects, we first cluster refactoring time series of the project stages (*i.e.,* in terms of WRD). Using the DTW algorithm, we measure the similarity between the WRD time series of each pair of project stages as part of the clustering process.

Clustering common refactoring practices. To identify refactoring tactics, we utilize WRD and form a time series that represent the refactoring history of each project. Subsequently, using DTW we cluster the projects based on the similarities of their refactoring activities represented by the time series. As the selected projects do not share a similar life cycle and they may experience different refactoring practices in different stages of development, we measure the similarities of refactoring activities between projects in different stages of development (*i.e.,* early, middle, late). Therefore, if a project has multiple development stages, we break its time series into multiple smaller time series, each of which represents one stage of the project.

We use Dynamic Time Warping (DTW), a clustering technique for temporal sequences based on their similarity, to cluster refactoring time series as refactoring tactics. We identify the optimal number of clusters using the elbow method [98]. The elbow method measures the sum of squared errors (SSE) and selects the smallest value

of k (*i.e.,* the number of clusters) with the lowest SSE as the optimal number of clusters. This is determined by identifying the point on the graph where the SSE begins to level off and form an elbow shape [99]. Moreover, we manually validate the optimal number of clusters (k) identified by the elbow method and check if our clustering results provide distinct centroids. Based on the elbow curve analysis depicted in Figure 3.6, four is identified as the optimal number of clusters. Furthermore, we utilize the silhouette score with the existing criteria [114–118] of the silhouette method to verify the optimal number of clusters. This identification is based on two conditions: (1) average silhouette score greater than 0.5 and (2) absence of clusters exhibiting all silhouette scores below the average. The silhouette score analysis points towards the optimal cluster numbers being 3 and 4, yielding average silhouette scores of 0.56 and 0.52, respectively. All clusters in both cases exhibit scores above the average threshold. This observation supports the idea that both 3 and 4 clusters could be considered as optimal solutions. However, as k=4 leads to a more even distribution of the sizes (*i.e.,* thicknesses) of the clusters [114], we opt for k=4. In summary, both the elbow curve and silhouette score analyses suggest that 4 clusters are the preferred number of clusters.

We utilize DTW to identify the similarities of refactoring time series by analyzing all stages of development in all projects together. Analyzing all stages together at the same time allows us to compare and identify the unified common behaviors in all stages of development despite their different life spans. By considering the optimal number of clusters as four and performing DTW, we identify four common behaviors based on the cluster centroids to represent the common tactics as variations of the *root canal* and *floss* refactoring tactics.

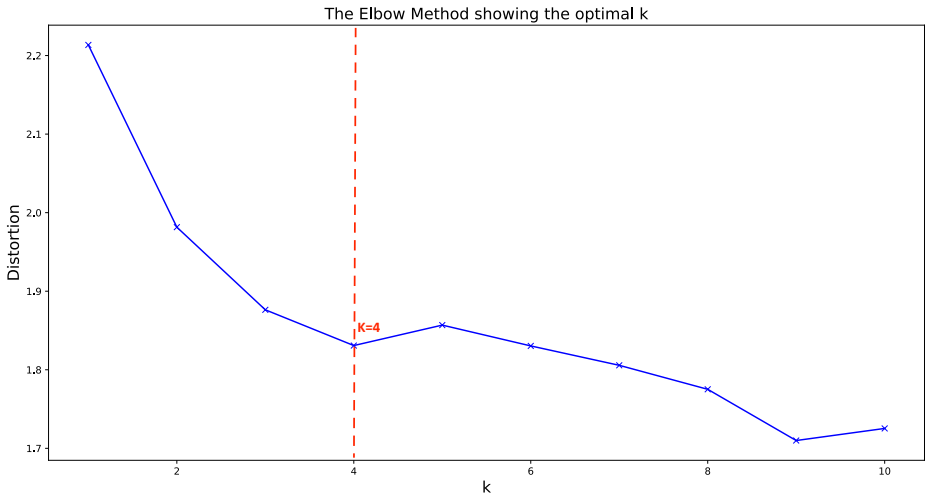


Figure 3.6: The results of the elbow curve, showing the optimal number of clusters using DTW.

Identifying refactoring spikes. To provide more insights on the identified tactics, we measure the number of spikes that happen in each refactoring tactic centroid. Our intuition is that a higher number of spikes within a refactoring tactic time series indicates more deviation of refactoring densities from regular refactoring densities. To determine a spike we apply the Median Absolute Deviation (MAD) method. Compared to the standard deviation, MAD is a robust estimator of scale. MAD can also be used as a scaling quantity instead of the standard deviation, which is vulnerable to the influence of extreme values [119, 120]. MAD can be calculated using the formula below where n is each data point and ñ is the median of all data points in a window:

$$MAD = median\left(|n - median(\tilde{n})|\right) \tag{3.4}$$

Therefore, to detect refactoring spikes, we iterate through the data points in our centroid time series within a window of four weeks (one month) before and after a given index, which represents a week of development. For each window, we calculate

the median absolute deviation (MAD). Then, we check if the absolute deviation of the data point at the current index from the median of the window is greater than three times the MAD [121]. If the condition is true, we consider it a refactoring spike.

Clustering refactoring tactics in terms of project and developer profiles. Using a similar approach to Section 3.3.1, we use the Scott-Knott-ESD [108, 109] to cluster the distribution of the project and author profiles associated with the refactoring tactics into statistically significant groups. We perform two separate clusterings for (1) for combinations of project profiles and refactoring tactics, and (2) for combinations of author profiles and refactoring tactics. Each cluster represents the distribution of project or author profiles in project stages corresponding to the identified tactic. For example, RC-Vibrant indicates the distribution of the *root canal* tactic across *vibrant* project stages. Hence, each group represents a statistically significant distribution of the project or author profiles associated with the refactoring tactics. The results of the Scott-Knott-ESD test reveal more insights into the identified refactoring tactics.

Findings

Our clustering approach uncovers four primary refactoring tactics. We define the intermittent spiked floss and the frequent spiked floss as two variations of the floss refactoring tactic, and the intermittent root canal and the frequent root canal as two variations of the root-canal refactoring tactic. The behavior (*i.e.,* changes in refactoring density over time) of refactoring tactics is illustrated in Figure 3.8. The main difference between the floss-based and root canal-based tactics is that: floss-based tactics mix refactoring with

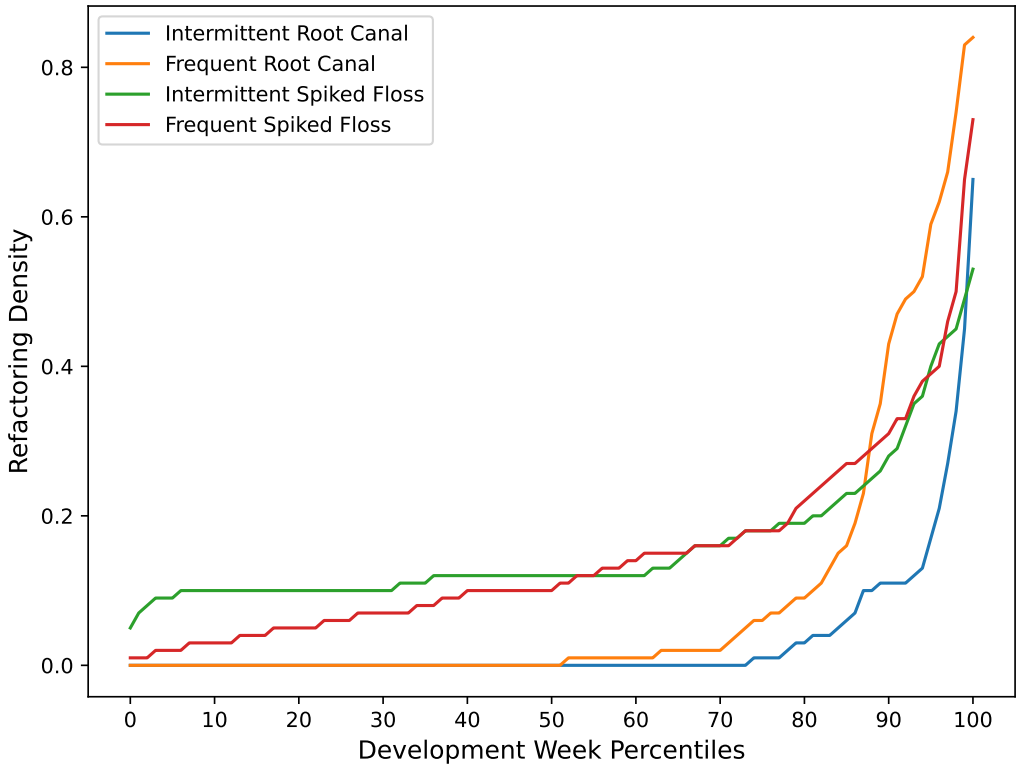


Figure 3.7: The relationship between development weeks percentiles and refactoring density.

Table 3.7: Summary of the number of refactoring spikes for each refactoring tactic centroid.

	Floss		**Root Canal**	
	Intermittent Spiked	**Frequent Spiked**	**Intermittent**	**Frequent**
Spikes Count	35	59	35	66

regular development activities (*i.e.,* the refactoring densities are consistently higher than zero, as indicated in Figure 3.8 (B and C)); in comparison, root canal-based tactics involve refactoring activities once in a while (*i.e.,* the refactoring densities are at or near zero for most of the time periods, as indicated in Figure 3.8 (A and D). Additionally, Figure 3.7 shows the refactoring density percentiles of root canal-based

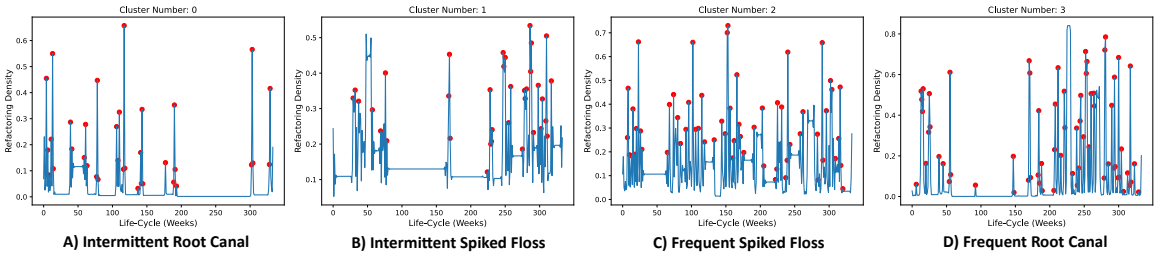


Figure 3.8: Clustering centroids that represent refactoring tactics identified in this study, which are labeled as: *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. The red dots show refactoring spikes in each tactic.

and floss-based tactics. As is shown root canal-based tactics have zero refactoring densities for more than half of the development cycle. Specifically, intermittent root canal have zero refactoring density until the 74^{th} percentile of development weeks (*i.e.*, more than 74% of the weeks have zero refactoring densities), and frequent root canal have zero refactoring until the 52^{nd} percentile (*i.e.,* more than 52% of the weeks have zero refactoring densities). In contrast, floss-based tactics have a non-zero median at all percentile points. Moreover, *frequent root canal* and *frequent floss* tend to have more frequent high-density periods than *intermittent spiked floss* and *intermittent root canal*. Furthermore, as it is shown in Table 3.7, by comparing refactoring spikes count in each tactic, we find that *intermittent spiked floss* has fewer spikes (35) compared to *frequent spiked floss* (59). Similarly, *frequent root canal* has more spikes (66) compared to *intermittent root canal*, which has 35 spikes. Overall, the *frequent root canal* and *frequent spiked floss* tactics exhibit more frequent high-density periods (*i.e.,* refactoring spikes) than the *intermittent root canal* and *intermittent spiked floss* tactics. We define each refactoring tactic as follows:

- **Intermittent spiked floss:** with refactoring consistently in all development

Table 3.8: The distribution of refactoring tactics in different stages of development

	Floss		**Root Canal**	
	Intermittent Spiked	**Frequent Spiked**	**Intermittent**	**Frequent**
Early	3%	52%	19%	26%
Middle	35%	12%	41%	12%
Late	21%	0%	78%	1%

weeks, developers perform refactoring on a regular basis along with fewer refactoring spikes compared to *frequent spiked floss.*

- **Frequent spiked floss:** with refactoring consistently in all development weeks, developers perform refactoring with more drops and increases (*i.e.,* spikes) in refactoring density compared to *intermittent spiked floss.*

- **Intermittent root canal:** with the majority of the weeks having zero refactoring densities, developers tend to perform refactoring irregularly but in high densities when they do perform it.

- **Frequent root canal:** with the majority of the weeks having zero refactoring densities, developers tend to perform more frequent refactorings with more spikes in refactoring density compared to *intermittent root canal.*

Software projects undergo different refactoring tactics during their lifetime. Table 3.8 shows the distribution of the identified tactics in different stages of development. In particular, in the early stage of development, the majority of refactoring tactics are floss-based (55%). This observation is aligned with the previous studies showing that the majority of refactoring tactics are floss-based refactoring [60, 122]. However, in the middle stage, the utilization of floss-based tactics drops to 47%. Finally, in the late stage of development, the majority of refactorings

are observed to be root canal-based tactics (79%). ***Therefore, the amount of floss-based refactoring tactics reduce while the projects enter their later stages of development: the developers aim for more targeted refactoring operations as the projects grow over time.*** Table 3.9 shows the results of the Scott-Knott-ESD test, which reveals more information on the distribution of refactoring tactics associated with different project and author profiles:

Among project profiles: In *maintaining* project stages, intermittent root canal (cluster 1) is frequently used (60%). While, in vibrant project stages, floss-based tactics (*frequent spiked floss* and *intermittent spiked floss*) (cluster 2) are more frequently used (79%). Moreover, the *obsolete* project stages mainly use intermittent root canal (cluster 2) and *growing* project stages (cluster 2) utilize *frequent spiked floss* and *intermittent root canal* as the main refactoring tactics. As Vibrant project stages have more and most contributors and commits, they have more active development activities; thus they are more likely to experience frequent refactoring during the development process (*i.e.,* floss-based tactics). However, *maintaining* and *obsolete* project stages with the least contributors try to maintain the code and keep it working by doing targeted refactoring from time to time.

Among author profiles: *casual* developers mainly do floss-based refactoring tactics (cluster 1 and 2) (*i.e., frequent spiked floss* and *intermittent spiked floss*), while *main* authors often utilize *frequent spiked floss* (cluster 1). Moreover, *core* authors utilize both *frequent spiked floss* and *intermittent root canal* (cluster 2). As *core* authors have the most and more contributions to the repository, they are more likely to contribute more to critical refactoring activities; therefore, they are more likely to do root canal-based refactoring, while letting *casual* contributors perform floss-based

Table 3.9: Scott-Knott-ESD test results on the refactoringtactics and associated project and author profiles.

Project Profiles			**Author Profiles**		
Tactic Profile	**Cluster Rank**	**Mean (%)**	**Tactic Profile**	**Cluster Rank**	**Mean (%)**
IR-Maintaining	1	0.60	FF-Casual	1	0.47
IR-Obsolet	2	0.42	FF-Main	1	0.42
FF-Growing	2	0.42	FF-Core	2	0.35
FF-Vibrant	2	0.41	IR-Core	2	0.31
IR-Growing	2	0.39	IF-Casual	2	0.30
IF-Vibrant	2	0.38	IF-Main	3	0.25
FF-Obsolete	3	0.30	IR-Main	3	0.24
FR-Maintaining	4	0.25	IR-Casual	4	0.19
FR-Obsolete	4	0.19	IF-Core	4	0.18
IR-Vibrant	4	0.18	FR-Core	4	0.16
FR-Growin	5	0.11	FR-Main	5	0.09
FF-Maintaining	5	0.09	FR-Casual	6	0.04
IF-Obsolete	5	0.09			
IF-Growing	5	0.08			
IF-Maintaining	6	0.06			
FR-Vibrant	6	0.03			

IR: intermittent root canal, IF: intermittent spiked floss, FF: frequent spiked floss, FR: frequent root canal

refactoring during the development process.

$RQ_{3.2}$ Summary

Apart from *floss* and *root canal* refactoring tactics, software developers use more diverse refactoring tactics, such as *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. Among project stages categorized as *obsolete* or *maintaining*, root canal-based tactics are prevalent, whereas floss-based tactics are more commonly employed in *vibrant* stages. Additionally, *core* authors tend to use more root canal-based tactics, whereas casual contributors are inclined towards floss-based tactics.

3.3.3 $RQ_{3.3}$: What is the relationship of different refactoring rhythms and tactics with code quality?

Motivation

In the first and second research questions, we identify different refactoring tactics and rhythms applied by different projects. Apart from finding different tactics and rhythms, identifying the relationship between refactoring tactics and code quality is crucial as it helps developers prioritize their efforts, improve development processes, and deliver high-quality software. Therefore, we utilize code smells as quality metrics for refactoring [9, 12] to compare the changes in quality after adopting each tactic or rhythm. Understanding the relationship of different rhythms and tactics with code quality can help practitioners and project teams to (1) discover the positive and negative aspects of the different refactoring rhythms or tactics, and (2) adopt or switch to the most suitable refactoring rhythm or tactic.

Approach

To understand the relationship of the identified refactoring rhythms and tactics with code quality, we utilize code smells listed in Table 3.1 as code quality metrics. Using Scott-Knott-ESD test [108,109], we cluster the magnitude of code smell changes (*i.e.,* increase/decrease) after adopting each tactic or rhythm. The Scott-Knott-ESD complements the Scott-Knott test [110] by taking the effect size difference into account when identifying different clusters. We first identify the relationship between the identified (1) refactoring tactics and (2) refactoring rhythms with overall increase or decrease in code smells as quality measures. Moreover, we identify the relationship of each refactoring tactic and rhythm with different types of code smells. We describe our detailed approach below.

Measuring code smell changes. As discussed in Section 3.2, to measure the relationship of the identified refactoring rhythms and tactics with code quality, we use code smells. We set three stages (early, middle, and late) for the lifetime of projects and then collect the code smell metrics, which are normalized by the project size, at the beginning and the end of each stage respectively.

The relationship of refactoring rhythms and tactics with the overall code quality. In Section 3.3.1, we classify refactoring rhythms as *all-day* and *work-day*. Besides, in Section 3.3.2, we identify four major refactoring tactics: *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. To identify the relationship between the above refactoring rhythms and tactics with code quality, we utilize the normalized frequency of code smell changes after adopting each rhythm and tactic (listed in Table 3.1). Therefore, a higher frequency of changes indicates an increase in code smells and a decrease in software quality. To analyze

the overall relationship of refactoring tactics and rhythms with the frequency of code smell changes, we use the normalized sum of all code smell changes as the overall changes in code smells and label them with the corresponding rhythm and tactic. Using the Scott-Knott-ESD test we cluster and rank the refactoring rhythms and tactics based on the code smell changes to identify the rhythms and tactics leading to more smelly code. We use $p - values < 0.05$ to identify the statistical significance and use means to rank the identified clusters.

The relationship of refactoring rhythms and tactics with each code smell type. To provide more insights and details on the identified rhythms and tactics, we conduct separate analyses to assess the relationship between the frequency of different types of code smells and each refactoring rhythm and tactic. We use 35 code smells (listed in Table 3.1) and the changes after adopting each rhythm and tactic. To this end, we utilize the Scott-Knott-ESD test to cluster ($p - values < 0.05$ as the significance threshold) and rank (using means) the rhythms and tactics based on each type of code smells separately. Therefore, we perform 35 individual tests for rhythms and 35 separate tests for tactics. Therefore, each Scott-Knott-ESD test is responsible for one type of code smells. Doing so allows us to identify how the refactoring rhythms and tactics impact each type of code smells.

Findings

In this section, we provide the findings on the relationship of both refactoring rhythms and tactics with code quality.

Overall relationship: We use the sum of all types of code smell changes (*i.e.,* the total number of code smell changes regardless of the code smell type) to measure the

overall code smell changes after adopting each refactoring rhythm and tactic. As the results suggest, the identified rhythms belong to the same cluster and do not significantly affect overall changes in the number of code smells (Table 3.11), however, the *all-day* refactoring rhythm is associated with the lowest mean in overall code smell changes, which indicates a higher code quality. Overall, refactoring rhythms are not statistically associated with the overall changes in the code smells. For refactoring tactics, on the other hand, *intermittent spiked floss* and *frequent spiked floss* are in the first and second ranked clusters, hence, they are associated with more increase in the overall changes of code smells compared to the *frequent root canal* and the *intermittent root canal* tactics. ***In fact, on average, floss-based tactics are associated with an increase in the frequency of code smells (positive mean as shown in Table 3.10), while root canal-based tactics are associated with a decrease in the frequency of code smells (negative mean as shown in Table 3.10).*** Therefore, root canal-based tactics (*i.e., frequent root canal* and *intermittent root canal*) are associated with a higher code quality compared to floss-based tactics. A possible explanation may be that floss-based refactoring is typically integrated with addressing daily maintenance tasks, such as bug fixes and the implementation of new features, while root canal-based refactoring focuses on improving the overall quality of the design. Figure 3.9 shows the results of the individual Scott-Knott tests applied for each type of code smell. We observe that, for 6% (2 out of 35 code smell types), namely empty catch clause and multifaceted abstraction, the different refactoring tactics are not associated with a statistically significant difference in the frequency of the corresponding code smell.

Relationship with specific code smells: The results from our analysis of the

Table 3.10: Scott-Knott-ESD test results on the overall changes in the frequency of code smells associated with the refactoring tactics.

	Floss		**Root Canal**	
	Intermittent Spiked	**Frequent Spiked**	**Intermittent**	**Frequent**
Cluster Rank	1	2	3	3
Mean	0.20	0.01	-0.03	-0.03

Table 3.11: Scott-Knott-ESD test results on the overall changes in the frequency of code smells associated with the refactoring rhythms.

	All Day	**Work Day**
Cluster Rank	1	1
Mean	0.01	0.02

overall changes in the number of code smells show a significant difference in the code smell changes between the floss-based and the root canal-based tactics. However, rhythms do not show a significant difference in the changes in the frequency of code smells. Therefore, we cluster the frequency of code smell changes in each code smell type after adopting each refactoring tactic separately. This was determined through the Scott-Knott-ESD tests resulting in a single cluster. However, root canal-based tactics result in statistically smaller increases in the frequency of code smells, indicating higher quality, for 80% (28 out of 35) of code smell types. This includes 90% of the implementation smell types (9 out of 10 types), 83% of the design smell types (15 out of 17 types), and 57% of the architecture smell types (4 out of 7 types). The five remaining code smell types (*i.e.,* ambiguous interface, scattered functionality, dense structure, imperative abstraction, and unnecessary abstraction) show slightly different clustering results (Figure 3.9) from the majority of the code smells (74%). Therefore, adopting root canal-based tactics results in the majority of improvements

Code Smell	Clusters			
Cyclic Dependency	IF(1)	FF(2)	IR(3)	FR(3)
God Component	IF(1)	FF(2)	IR(3)	FR(3)
Ambiguous Interface	FF(1)	FR(1)	IF(1)	IR(2)
Feature Concentration	IF(1)	FF(2)	FR(3)	IR(3)
Unstable Dependency	IF(1)	FF(2)	IR(3)	FR(3)
Scattered Functionality	IR(1)	IF(1)	FF(1)	FR(1)
Dense Structure	IF(1)	FF(2)	IR(2)	FR(2)
Imperative Abstraction	IF(1)	FR(2)	FF(2)	IR(3)
Multifaceted Abstraction	IF(1)	FR(1)	FF(1)	IR(1)
Unnecessary Abstraction	IF(1)	FF(1)	FR(1)	IR(2)
Unutilized Abstraction	IF(1)	FF(1)	FR(2)	IR(2)
Feature Envy	IF(1)	FF(1)	FR(2)	IR(2)
Deficient Encapsulation	IF(1)	FF(2)	IR(3)	FR(3)
Unexploited Encapsulation	IF(1)	FF(2)	FR(3)	IR(3)
Broken Modularization	FF(1)	IF(1)	FR(2)	IR(2)
Cyclically Dependent Modularization	IF(1)	FF(2)	IR(3)	FR(3)
Hub-like Modularization	IF(1)	FF(2)	IR(3)	FR(3)
Insufficient Modularization	IF(1)	FF(2)	IR(3)	FR(3)
Broken Hierarchy	IF(1)	FF(2)	FR(3)	IR(3)
Cyclic Hierarchy	IF(1)	FF(2)	IR(3)	FR(3)
Deep Hierarchy	FF(1)	IF(1)	IR(2)	FR(2)
Missing Hierarchy	IF(1)	FF(2)	FR(3)	IR(3)
Multipath Hierarchy	IF(1)	FF(1)	IR(2)	FR(3)
Rebellious Hierarchy	FF(1)	IF(2)	IR(3)	FR(3)
Wide Hierarchy	IF(1)	FF(2)	IR(3)	FR(3)
Abstract Function Call from Constructor	IF(1)	FF(2)	FR(3)	IR(3)
Complex Conditional	IF(1)	FF(2)	FR(3)	IR(3)
Complex Method	IF(1)	FF(2)	IR(3)	FR(3)
Empty Catch Clause	IF(1)	FF(1)	FR(1)	IR(1)
Long Identifier	IF(1)	FF(2)	IR(3)	FR(4)
Long Method	IF(1)	FF(2)	IR(3)	FR(3)
Long Parameter List	IF(1)	FF(2)	IR(3)	FR(3)
Long Statement	IF(1)	FF(2)	FR(3)	IR(3)
Magic Number	IF(1)	FF(2)	IR(3)	FR(3)
Missing Default	IF(1)	FF(2)	IR(3)	FR(3)
Overall	IF(1)	FF(2)	IR(3)	FR(3)

Intermittent Spiked Floss | Frequent Spiked Floss | Intermittent Root Canal | Frequent Root Canal

Figure 3.9: Results from the Scott-Knott-ESD tests that cluster and rank the refactoring tactics for overall and different types of code smell changes. A higher rank indicates a larger increase (or smaller decrease) in code smells.

in code smells across all three categories (*i.e.,* implementation, design, and architecture smells) of code smells. Overall, our results suggest that more dedicated refactoring efforts (i.e., using root canal-based tactics) can better help remove or fix most types of code smells.

$RQ_{3.3}$ Summary

Root canal-based tactics are associated with a greater decrease (or smaller increase) in the number of code smells, and thus higher code quality, compared to floss-based tactics, which suggests more dedicated refactoring operations. However, refactoring rhythms are not associated with the changes in the number of code smells, suggesting that the choice of rhythm may be driven more by project-specific factors and team preferences rather than their impact on code quality.

3.4 Threats to Validity

In this section, we discuss the possible threats to the validity of our study.

Internal validity. Concerning our project selection and selected approaches, in the second research question, for clustering refactoring time series and finding refactoring tactics, we analyze the refactoring densities in three stages of development. We choose the mentioned time frames so that we could compare the refactoring behaviors of the project stages with similar length of development history. We admit having more projects with different lengths of time frames could reveal more refactoring tactics. Moreover, due to the varying lengths of the life cycles of projects in stages after the late refactoring stage, time series clustering could not be applied, and we had to exclude them from our study. Thus, it is possible that some patterns may emerge in later stages that we were unable to capture. In the second research question, We

have categorized the data into four distinct clusters, namely *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. The number of clusters chosen may impact the quality and comprehensibility of the clustering outcomes, as well as the insights and conclusions derived from them. If there are too many clusters, it may result in overfitting, whereas if there are too few, important information may be lost. To avoid bias, we use the elbow method [98], silhouette score [118], and manual inspection to identify the optimum number of clusters. However, different numbers of clusters could reveal less or more refactoring tactics. In the third research question, we use code smells as a code quality measure to study the relationship between refactoring rhythms or tactics with code quality. Nevertheless, we agree that code quality can be characterized by other measures, such as the number of bugs or maintenance costs. Furthermore, we admit that other socio-technical metrics, such as the way refactoring is applied (*e.g.,* manually or automatically) and regulations of the development team could affect our code quality measurement. Future work that explores the relationship between refactoring activities and other characteristics of code quality could complement our results.

External validity. Concerning the generalization of our findings, our experiments and results are based solely on the analysis of the 196 Apache projects we studied, and therefore, our conclusions may not necessarily apply to other projects, such as those in different domains. Additionally, since our analysis was limited to projects written in Java, the findings may not be applicable to projects written in other programming languages.

Construct validity. Concerning our measurement accuracy, in the third research question, to study the relationship between refactoring rhythms and code quality

(*i.e.,* in terms of the frequency of code smell changes), we measure the code smells in different stages of the projects, because calculating code smells every week takes approximately 25 days for each project, and computing them for all projects requires a significant amount of time. Nevertheless, extracting quality changes every week could provide more accurate results.

3.5 Summary

In this chapter, we investigate the refactoring activities on a dataset consisting of 196 Apache projects to identify refactoring tactics and rhythms that developers and projects adopt in the software development process. We also examine their relationship with code quality in terms of code smells. Comparing both refactoring and development activities, we first determine that in more than 95% of project stages developers use a systematic refactoring rhythm on weekdays. Two major rhythms are identified as 1) *work-day* refactoring and (2) *all-day* refactoring. By considering the relationship between refactoring rhythms and the quality metrics (*i.e.*, code smells), we observe that different refactoring rhythms do not make a statistically significant difference to the code quality. Moreover, by clustering the life-cycle of refactoring activities we find four variations of existing refactoring tactics: (1) *frequent spiked floss*, (2) *intermittent spiked floss*, (3) *frequent root canal*, and (4) *intermittent root canal* refactoring tactics. We observe that root canal-based tactics (*frequent root canal* and *intermittent root canal*) are associated with a larger reduction in the frequency of code smells compared to floss-based tactics (*frequent spiked floss* and *intermittent spiked floss*). Our findings can help researchers and practitioners understand practical refactoring activities in real-world projects and their relationship with code quality.

Practitioners can leverage our findings to choose the appropriate refactoring patterns for their projects based on their resources and code quality requirements. For future work, we plan to conduct experiments for other programming languages and focus more on automatic vs. manual refactoring operations.

Chapter 4

A Study on Release-Wise Refactoring Patterns

This chapter describes our study on refactoring patterns within release cycles. Section 4.1 explains the research problem and motivation of our study. Section 4.2 provides the experiment setup of our study. Section 4.3 presents the motivation, approaches, and findings of our research questions. Section 4.4 explains the threats to the validity of our findings. Finally, we conclude our study and present future research directions in Section 4.5.

4.1 Problem and Motivation

Chapter 3 explores different variations of refactoring tactics, which describe how developers perform refactoring over the long term and correlates them with code quality. With the increased utilization of continuous integration and continuous deployment (CI/CD) [45], software projects now frequently release updates in shorter cycles [46,47]. These continuous rapid release cycles, which can span from a few days to several weeks [49–51], place pressure on developers to incorporate user feedback and release a new version. However, refactoring strategies applied in short-term release cycles have not yet been thoroughly explored. Given the substantial amount of

effort developers invest in refactoring [14,15], it remains unclear how these efforts are distributed throughout release cycles. Shorter release cycles may prompt the adoption of specific refactoring patterns aligned with release dates. This chapter aims to address this gap by analyzing refactoring strategies within short-term release cycles and identifying patterns that contribute to high-quality software delivery.

In this chapter, we conduct a large-scale empirical analysis of refactoring practices across 1,604 releases from 207 open-source projects. We define **release-wise refactoring patterns** to describe *how practitioners distribute their refactoring activities throughout a release*. Our goal is to identify and analyze the dominant release-wise refactoring patterns by examining changes in refactoring density throughout each release. Additionally, we assess the relationship between release-wise refactoring patterns and code quality by measuring code smells and metrics (*e.g.,* cohesion and coupling) before and after each release. Based on our findings, we aim to propose best practices for refactoring within a release, including strategies for switching between different patterns and their effects on code quality.

We aim to answer the following research questions:

***RQ*$_{4.1}$**: What release-wise refactoring patterns are present in open-source projects?

> Developers spend effort on refactoring during the software development process; however, it is not clear how they allocate refactoring tasks within a release. We study the evolution of refactoring densities throughout software releases and identify four major release-wise refactoring patterns: *early active*, *late active*, *steady active*, and *steady inactive*. Furthermore, by analyzing the types of refactoring and external factors (*e.g.,* the number of developers) associated with each release, we provide details of the

characteristics of each release-wise refactoring pattern.

***RQ*$_{4.2}$**: RQ2. What is the relationship between release-wise refactoring patterns and code quality?

To identify the most effective refactoring patterns, we assess the relationship between release-wise refactoring patterns and code quality. We use code smells and code metrics as code quality indicators to evaluate the quality of the code before and after each release. Then, we rank and cluster the identified release-wise refactoring patterns based on improvements in these quality metrics. Our findings indicate that *late active* refactoring patterns are associated with a greater reduction in code smells and higher code quality compared to the other identified patterns.

***RQ*$_{4.3}$**: RQ3. Does the usage of release-wise refactoring patterns change over time, and how do developers switch from one pattern to another?

To understand how practitioners' refactoring patterns evolve across the lifecycle of a project, we analyze the distribution of various release-wise refactoring patterns during the early, middle, late, and last stages of the project lifecycles, as well as the transitions among these patterns. We find that developers tend to continue using previous refactoring patterns. Moreover, we observe an increase in the utilization of the *steady active* pattern, while the adoption of the *steady inactive* pattern decreases over time. Additionally, the usage of the *late active* refactoring pattern remains consistent across all development stages. While *early active* and *steady active* patterns may offer temporarily improved release quality, the

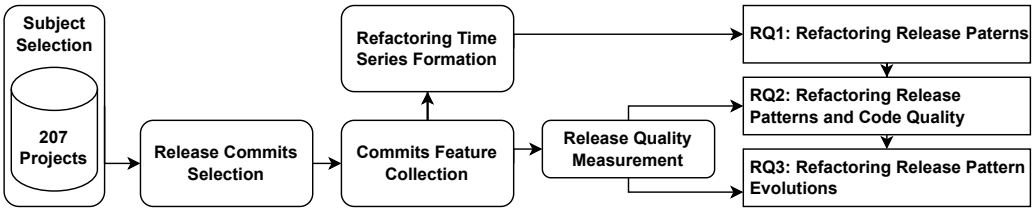


Figure 4.1: The overview of our study.

steady inactive pattern contributes to higher code quality when applied continuously.

The replication package for this chapter is available online[1].

4.2 Experiment Setup

This section details the experimental setup of our study, covering our methods for data collection, pre-processing, and analysis.

4.2.1 Overview of Our Approach

An overview of our study is shown in Figure 4.1. We conduct our experiments by systematically selecting 207 active and popular open-source Java projects with sufficient commit history. Furthermore, we select the major and minor releases within each project and identify the responsible commit for each release. From the selected commits, we extract code and refactoring-related features as well as project-related features such as the number of developers. We then create a time series describing the refactoring changes within each release. We answer the first research question by clustering the release-wise refactoring time series and identifying the main patterns therewithin. By measuring code smells and code metrics as code quality indicators

[1] https://github.com/Shayan-Noei-PhD-Replication-Packages/Chapter-4-Release-Wise

before and after each release, we explain the best release-wise refactoring patterns for maintaining code quality, addressing the second research question. Finally, by analyzing the distribution of the identified release-wise refactoring patterns across different stages of the development lifecycle and examining the transition among them, we aim to understand how developers use and switch between these patterns and their implications for code quality. We detail the analyses specific to each individual research question in Section 4.3.

4.2.2 Subject Selection

We use a systematic approach to select the subject projects for our study. Considering Java is one of the most popular programming languages [24, 123], and it is supported by robust refactoring detection tools, such as RMiner [10, 21], we start by collecting all Java repositories with at least one star and one fork using the GitHub advanced search[2], which yields 490,880 Java repositories on GitHub. To ensure we include the most active and popular projects with a sufficient history of development, we select repositories that:

- are up to date with current development practices
  - have been active in the year of the data collection (with at least one push in 2023)
- are not disabled or archived
- have a healthy level of interest and activity in the repository

[2]https://github.com/search/advanced

  - have more than 13 forks[3] (13 is calculated as the 3rd quartile of the distribution of number of forks of all the projects)

- are popular within the open-source community
  - have more than 22 stars (22 is calculated as the 3rd quartile of the number of stars)
- have a sufficient development history for our analysis
  - have at least 1,000 commits, similar to prior work [124]
- have enough releases to conduct our experiments (at least two releases), and
- provides sufficient data on the details of their releases
  - follow a standard semantic versioning format (*e.g.,* v2.0.3), which consists of three numbers in the format MAJOR.MINOR.PATCH [125]

The different types of releases are described below:

- *Major release:* focuses on introducing significant new features or architectural changes to the software. The versioning convention for a major release is a number followed by two zeroes (*e.g.,* v1.0.0).
- *Minor release:* is typically used to introduce a new feature or a minor improvement in the software. It is represented by the major version number followed by the minor version number and a zero (*e.g.,* v1.2.0).

[3] A copy of that repository under a different GitHub account.

- *Patch release:* is often used for a hotfix or a minor improvement in the software. It is indicated by the major version number followed by the minor version number and the patch number (*e.g.,* v1.2.3).

As a result, we select 207 repositories as the subject dataset for our study.

4.2.3 Release Commits Selection

Different projects choose different release strategies to release their software [126]. Some projects may choose a mainline branch to release, while others may choose different branches for their feature development or releases [127]. The projects in our dataset utilize the following dominant release strategies:

- **Mainline release strategy:** This strategy is a single-branch release strategy, in which the project uses only one branch for releases and keeps developments in a separate branch. Consequently, all releases originate from the main/master branch. 121 projects in our dataset mainly employ the *mainline* release strategy.

- **Release branch strategy:** This strategy focuses on creating a dedicated branch for each release and releasing from that specific branch. As a result, each software release has its own branch, and all versions are accessible through their respective branches. 86 projects of our dataset mainly follow the *release branch* strategy.

As we aim to analyze the refactoring patterns corresponding to each release, we focus on major and minor releases as they typically reflect significant changes and improvements. Patch releases, on the other hand, usually address single hotfixes and may not contribute to release-wise refactoring pattern analysis. To select the

commits corresponding to each major/minor release, we retrieve release information from the GitHub REST API[4] for each repository to collect their major/minor release dates. Depending on the release strategy employed by the projects, we first identify the branch used for the release and then select the commits with timestamps that fall within the relevant release period from that branch, and then sort these commits based on their timestamps. We identify release strategies by analyzing the branches from which releases are made. If a project consistently uses the master/main branch, we identify it as following a mainline strategy. Projects consistently using different branches for releases are classified under the release branch strategy. If a project releases from the master/main branch and then switches to a different branch, we classify it as using both techniques.

4.2.4 Commit Feature Extraction

For each commit, we select (1) the commit log information that includes details such as the author and commit date, and (2) the code change and refactoring information that contains the files changed, lines of code changes, and the frequency and types of refactoring applied in each commit. To obtain the refactoring information, we use RMiner 3.0.0, which is the most accurate state-of-the-art refactoring detection with the maximum number of refactoring types (102 types) detected [124, 128]. RMiner has demonstrated a precision of 99.7% and a recall of 94.2% [10]. This approach enables us to capture not only basic commit information but also details such as the affected files, refactoring lines, and refactoring operations (*e.g.,* pull-up method).

[4]https://docs.github.com/en/rest

4.2.5 Code Quality Measurement

Code smells have been shown as effective refactoring quality indicators, as refactorings tend to eliminate code smells [34, 41, 124]. To this end, we use Designite 2.5.5 [129, 130] to measure code smells before and after each release. Designite is a robust tool that surpasses its competitors, capable of detecting 47 types of code smells across various code levels, which fall into five higher-level code smell categories:

- **Architecture smells:** explain the issues related to overall system structure and dependencies, such as cyclic dependencies. This category contains 7 code smell types.
- **Design smells:** include problems related to fundamental design principles, such as unnecessary abstractions and deficient encapsulation. This category contains 18 code smell types.
- **Testability smells:** explain the aspects of testability of the code, such as excessive dependency. This category includes 4 code smell types.
- **Implementation smells:** explain concerns with code implementation, like complex methods, long parameter lists, or long statements. This category has 10 code smell types.
- **Test smells:** include the problems within the test code, such as missing assertions, and ignored tests. This category includes 8 code smell types.

If refactoring is not performed properly, it can negatively impact the overall quality of the code. Previous studies have shown that refactoring is not always motivated by the elimination of code smells [40]. Therefore, we consider common code quality

metrics such as cohesion, coupling, and complexity [131] to measure code quality before and after refactoring. To achieve this, we use the Understand tool [132], which is capable of identifying 42 code metrics within the Java codebase. We group these metrics into three main categories: lack of cohesion, coupling, and complexity. The explanation of each category is as follows:

- **Lack of Cohesion**: Describe the extent to which functions and classes of the code work together. We measure this by evaluating the lack of cohesion among functions and classes.
- **Coupling**: This category of metrics describes the degree of interdependence between different classes and functions of the code. We assess coupling by examining the number of base classes, coupled classes, derived classes, and the inputs and outputs of functions and classes.
- **Complexity**: Describe the complexity of the code components. We assess complexity by analyzing cyclomatic complexity [133] and the max nesting size of statements (*e.g.,* for loops inside while loops).

4.3 Results

In this section, we discuss the motivation, approach, and findings for each of our research questions.

4.3.1 $RQ_{4.1}$: What release-wise refactoring patterns are present in open-source projects?

Motivation

Previous studies have examined long-term refactoring strategies, categorizing them into two general types: floss and root canal refactorings, with some variations in the frequency of their applications [5,44,124]. Additionally, these studies have identified a correlation between adopting different refactoring strategies and their impact on code quality [43, 124]. With the increased adoption of continuous integration and continuous development (CI/CD) in recent years [134], development and feature deliveries are segmented into short release cycles However, there is no large-scale study on the refactoring strategies relative to release cycles and their impact on code quality. In this research question, we aim to identify the frequent patterns developers utilize to integrate refactoring tasks within software releases. By identifying these patterns, we aim to deepen our understanding of refactoring practices and measure their efficacy in the fast-release cycles. Ultimately, we aim to identify the most successful release-wise refactoring strategies to improve code quality and establish best refactoring practices in CI/CD development.

Approach

To identify and gain insights into release-wise refactoring patterns, we first measure refactoring density over time, then form a normalized refactoring time series for each release. Finally, using time series clustering, we identify common refactoring patterns and their characteristics within the releases. In the following, we provide detailed explanations of each step conducted in this process:

Calculating refactoring density: To identify the refactoring trends and study changes in refactoring activities within each release, we use a metric to capture the density of daily refactoring activities with each release. Following prior work [124], we use the commits submitted on each active day of development to calculate the daily refactoring density (DRD), as defined in Equation 4.1.

$$\text{DRD}(i) = \frac{\text{Total refactoring churn of the day } (i)}{\text{Total code churn of the day } (i)} \tag{4.1}$$

DRD explains the amount of refactoring effort that is deviated from the regular development process. Using the DRD metric within each release, we create the refactoring-release time series, which represents the changes in refactoring density within each release.

Normalizing release time series: Due to differences in the length of release cycles, the refactoring time series of different releases vary in length, making them incomparable, as they cannot be directly mapped onto each other. To identify common release-wise refactoring patterns, we standardize the created release-wise refactoring time series within each release using linear interpolation [135, 136], which fits time series of different releases into a fixed range of 10 data points representing 10 quartiles of refactoring densities within each release. Linear interpolation estimates the DRD at a desired point based on the values of two neighboring points on the line connecting them [137]. Therefore, linear interpolation does not alter the overall trend but scales the time series into the same range. To compare historical refactoring information across releases and better interpret release-wise patterns, we calculate DRD for the entire refactoring history of before each release. This historical refactoring density provides insight into the amount of past refactoring effort from the beginning of the

project until the start of a release.

Identifying refactoring release patterns: To analyze trend patterns applied within a release, we utilize Soft Dynamic Time Warping [138] (Soft-DTW) for clustering our time series of refactoring intensities (*i.e.,* DRDs). Each cluster represents a release-wise refactoring pattern, specifically reflecting common refactoring activities within each release. Soft-DTW is an extension of DTW designed for time series with variable speeds, enabling it to capture the overall pattern trends of refactoring within a release. It allows for adjustments in time series that may begin their trends slightly earlier or later. Soft-DTW helps to minimize the impact of noise in our release-wise refactoring time series (*i.e.,* random fluctuations or outliers in refactoring time series) by incorporating a smoothing function that is particularly effective in handling noisy data. This function smooths the time series, reducing sensitivity to noise during clustering without altering the original data. To determine the optimal number of clusters, we first plot t-distributed Stochastic Neighbor Embedding (t-SNE) [139] and calculate the silhouette score [118] for different numbers of clusters. The t-SNE plots help visualize the distribution of time series in a reduced 2D space, while the silhouette score, ranging from -1 to 1, assesses the quality of the clustering. Then, we manually examine the cluster centroids to ensure they represent distinct and meaningful patterns.

Analyzing the release-wise refactoring patterns: Given that RMiner [10,21] can identify 102 distinct types of refactorings, we categorize these refactorings into six levels of granularity: variable, method, class, package, organization, and test levels. The description of each granularity is below:

- **Variable:** Focuses on variable levels within functions or classes (*e.g.,* , rename

variable).

- **Method:** Contains refactoring at the method level of the code. For example, extracting smaller methods from a larger one (*i.e.,* , extract method).
- **Class:** Consists of refactorings that involve a class. For instance, extracting a superclass.
- **Package:** Includes refactorings that involve multiple classes grouped as a package. For example, merging multiple packages into one.
- **Organization:** Contains refactorings that address the overall structure of the code. For example, moving code within files.
- **Test:** Has refactorings for test files. For example, parameterized tests, which involve running tests with different sets of input data.

To characterize the changes in the refactoring operations applied during the time span of a release, we partition each release into several development phases. Specifically, we divide each release into three time segments using the first and third quartiles of the release duration, representing the early (before the first quartile), middle (between the first and third quartiles), and late (after the third quartile) phases of the release cycle. We are interested in determining how refactoring activities evolve across these different phases and the corresponding refactoring categories. To this end, we perform a Kruskal-Wallis [101] test for each refactoring category (*e.g.,* method level) within each release-wise refactoring pattern to identify if there is a significant difference in the adoption of various refactoring operations across different phases of a release.

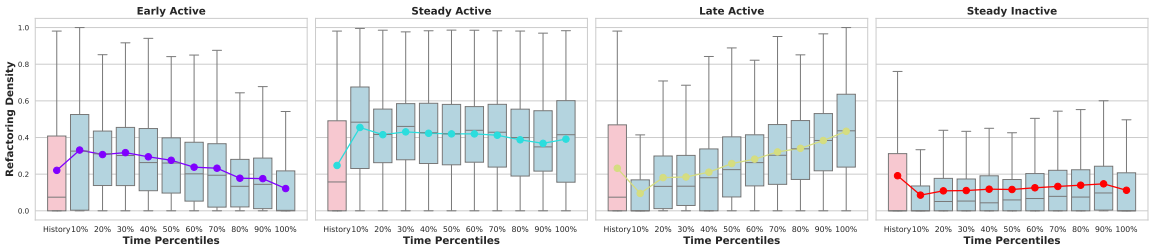


Figure 4.2: Boxplot of release-wise refactoring patterns. The dotted line represents the centroid of each cluster. The blue bars indicate the refactoring density of the current release and the pink bar indicates the historical refactoring density that shows the distribution of refactoring density from the start of the project up to the current release.

Analyzing the external features: To gain a contextual understanding of the external features contributing to the adoption of each release-wise refactoring pattern, we analyze three features of: project size (*i.e.,* code size), development community size (*i.e.,* the number of contributors), and project popularity (*i.e.,* the number of stars)—thereby providing a clearer picture of how these external features relate to different refactoring patterns. We divide each feature into four equal parts and label them as *least*, *less*, *more*, and *most* [124] (*i.e.,* a release with the most contributors). We analyze each categorized feature using the chi-square test [140], which determines if there is a significant difference (p-value ≤ 0.05) between the categorized values of external features across different release-wise refactoring patterns. Additionally, we present the distribution of each feature for each refactoring pattern.

Results

From the total of 1,604 analyzed releases, our clustering approach reveals four dominant refactoring patterns relative to software releases across all releases: (1) early active (30%), (2) steady active (17%), (3)

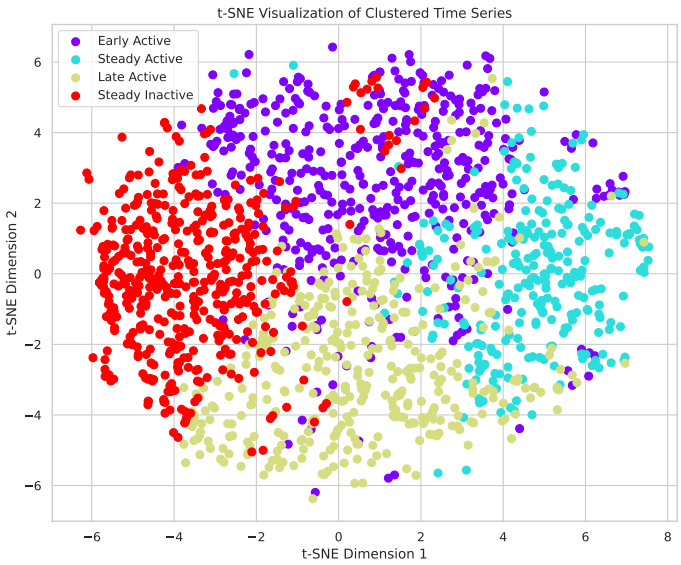


Figure 4.3: t-SNE plot showing the distribution of our clustering results in two dimensions.

late active (23%), and (4) steady inactive (30%). Figure 4.2 illustrates the evolution of each identified pattern within a release. As shown in Figure 4.2, the *early active* pattern shows a higher refactoring density at the beginning of the release, and then it gradually decreases as it gets closer to the release. In contrast, the *late active* pattern exhibits patterns with an increased refactoring density as it approaches the release. The *steady active* pattern indicates a consistently higher density of refactoring compared to the average history, while the *steady inactive* shows a low density of refactoring throughout the release. Interestingly, more than half of the project releases (53%) exhibit varying (increasing/decreasing) refactoring intensities in their release cycles. Figure 4.3 shows the t-SNE plot of the identified clusters of our release-wise time series. The plot indicates a clear separation between clusters, highlighting distinct patterns within the data. Furthermore, our approach yields a

Table 4.1: Median of the sum of distributions for different refactoring types in release-wise refactoring patterns per 1,000 lines of code churn, where significant differences in distributions are observed at different phases of the release. Non-existing categories do not show significant differences across phases and are thus applied equally.

Pattern Name	Category	Early	Middle	Late	P-value
Early Active	Variable	20.06	14.93	9.41	0.000
	Method	61.13	53.17	37.91	0.000
	Class	20.60	16.32	11.82	0.005
	All	126.13	103.83	83.69	0.000
Late Active	Variable	11.93	18.35	18.58	0.001
	Method	38.41	55.31	60.06	0.001
	All	86.78	109.5	119.98	0.002
Steady Active	Variable	22.53	20.07	15.49	0.011
	All	122.35	93.19	96.54	0.032

silhouette score of 0.3, suggesting an acceptable separation of the data.

Early active releases tend to have more class/method/variable-level refactoring earlier in a release, whereas late active releases tend to avoid extensive class-level refactoring late in a release. We use the Kruskal-Wallis test [101] to examine the differences in refactoring granularities across various phases of development with the following hypothesis:

- ***H_0:*** *There is no significant difference in the utilization of different granularities of refactorings across different phases of each release-wise refactoring pattern.*

Table 4.1 shows the results of comparing the distribution of different refactoring types across the early, middle, and late phases of development for each pattern, for the cases where the H_0 hypothesis is rejected (p-value ≤ 0.05). As it is shown, ***the adoption of package, organization, and testing level refactorings does not exhibit statistically significant differences across any phases of the releases in various patterns.*** Furthermore, in the *early active* pattern, with lower

Figure 4.4: Distribution of Features Across Refactoring Release Patterns.

DRD at the beginning of the release, there is an increase in refactoring activities at the method, class, and variable levels initially, which then significantly decreases closer to the release. Conversely, in late active, method, and variable level refactorings significantly increase as the release approaches, while other categories, including the class level refactoring, remain at the same pace. In the *steady inactive* pattern, there are no significant variations in overall refactoring types, indicating a consistent low frequency of refactoring throughout the release cycle. Likewise, the *steady active* pattern experiences a consistently increased frequency of refactoring throughout the release, except for variable-level refactoring operations, which are considered lower-level refactorings.

Projects utilizing different release-wise refactoring patterns exhibit distinct characteristics. Figure 4.4 shows the distribution of size, contributors, and stars external features among different release-wise refactoring patterns. The results of the chi-square test indicate significant differences in the distribution of each external feature among different refactoring release patterns, with p-values of 0.003 for size, 0.000 for contributors, and 0.001 for stars, respectively. We observe that:

- The *steady inactive* pattern is associated with *less* size, *less* developers, and the *most* stars.
- The *steady active* pattern is associated with *least* size, the *least* developers, and *less* stars.
- The *early active* pattern is associated with the *more* size, the *most* developers, and *more* stars.
- The late active pattern shows the *most* size, *more* developers, and *least* stars.

Smaller projects with fewer developers are mainly associated with either steady active or steady inactive patterns, where refactoring activities remain consistent throughout a release. In contrast, larger projects with more developer numbers exhibit a variety of refactoring patterns, where refactoring activities may increase or decrease over time. This suggests that smaller projects tend to integrate refactoring into daily development activities, while larger projects may concentrate refactoring efforts in specific phases of a release, such as the early or late stages.

$RQ_{4.1}$ Summary

We identify four dominant release-wise refactoring patterns: *late active*, *early active*, *steady active*, and *steady inactive*, each associated with different types of refactoring and project characteristics.

4.3.2 $RQ_{4.2}$: What is the relationship between release-wise refactoring patterns and code quality?

Motivation

The first research question identifies four primary refactoring patterns developers utilize. Understanding the relationship of these patterns with code quality can help developers make informed decisions to manage and distribute refactoring tasks throughout the release cycle. With the understanding, we can provide practitioners with insights into how adopting different release-wise refactoring patterns could potentially increase code quality. In this research question, we investigate the relationship between release-wise refactoring patterns and code quality measured by code smells and quality-related code metrics.

Approach

To measure the relationship between the detected release-wise refactoring patterns, we first use a set of code quality metrics to assess changes in quality after applying each pattern. We then rank and cluster the refactoring patterns based on these quality metric changes to evaluate their effectiveness. The following provides a detailed explanation of each step in this process:

Quality measurement: Previous studies have identified code smells as indicators of refactoring quality [34, 41, 124]. However, refactoring may not be motivated solely by the elimination of code smells [40]. Therefore, we measure the effectiveness of the release-wise refactoring patterns in eliminating or reducing code smells and use changes in code metrics (*e.g.,* low coupling) to complement these results. This provides additional details on the overall quality improvements of the code after

applying each refactoring pattern. We measure changes in code smells categorized into six granularities and code metrics categorized into three granularities (as explained in Section 4.2.5), before and after each release. We then use the following equation to measure the normalized overall and category-specific differences in code smells/metrics (CSD) before and after each release [124]:

$$\mathrm{CSD}(i) = \frac{(\mathrm{ECS}(i)/\mathrm{ELC}(i) - \mathrm{ICS}(i)/\mathrm{ILC}(i))}{\mathrm{CC}(i)/\mathrm{ELC}(i)} \tag{4.2}$$

ECS indicates the frequency of code smells/metrics at the end of a release; ELC indicates the lines of code at the end of a release; ICS shows the initial code smells/metrics of a release; ILC indicates the initial lines of code at the beginning of a release; and CC indicates the total code churn of each release. This equation calculates the change in code smell per line from the start to the change in code smell at the end of a release, normalized by the code churn relative to the codebase size, providing a measure of how code quality has been impacted by modifications during that release.

Ranking and clustering: To measure similarities and rank the identified release-wise refactoring patterns, we use the Scott-Knott-ESD [108, 109] test. This test helps us: (1) perform statistical comparisons among the metric value changes of different patterns and rank the release-wise refactoring patterns based on the magnitude of quality changes, and (2) cluster release-wise refactoring patterns that do not exhibit significant differences in quality measures. Therefore, we can identify which groups of release-wise refactoring patterns do not show significant differences and which group represents the most effective treatment (*i.e.,* improvement in each quality measure).

Table 4.2: The results of the Scott-Knott-ESD test on **code metrics** within each release-wise refactoring pattern. Patterns within the same cluster are represented by the same color. Clusters shown in red indicate lower quality compared to those highlighted in green.

Complexity			**Coupling**			**Lack of Cohesion**		
Pattern	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**
Steady Inactive	1	2.51	Steady Inactive	1	6.85	Steady Inactive	1	5.83
Early Active	1	1.77	Early Active	1	4.59	Early Active	1	3.70
Steady Active	1	1.54	Late Active	2	4.13	Late Active	2	3.41
Late Active	1	1.53	Steady Active	2	3.54	Steady Active	2	3.39

Findings

Late active and steady active release-wise refactoring patterns exhibit higher quality by improving cohesion and reducing coupling. Overall lower coupling and higher cohesion (*i.e.,* lower lack of cohesion) contribute to better code quality. As shown in Table 4.2, *steady inactive* and *early active* release-wise refactoring patterns have higher mean values and ranks (*i.e.,* rank 1) for coupling, indicating higher coupling, and higher mean and rank in lack of cohesion, indicating lower cohesion, compared to *late active* and *steady active* release-wise refactoring patterns. Therefore, *steady inactive* and *early active* release-wise refactoring patterns exhibit lower code quality compared to *late active* and *steady active* patterns. The *steady ease* and *early ease* release-wise refactoring patterns may indicate little or no refactoring, with a focus on addressing previous developments rather than the current release. In contrast, the *steady active* and *late active* patterns reflect a more cleaning-focused phase before or throughout the release, likely targeting the current state of the code in preparation for the release.

The steady inactive release-wise refactoring pattern exhibits the worst performance in design, implementation, and total code smells reduction.

Table 4.3: The results of the Scott-Knott-ESD test on **code smells** within each release-wise refactoring pattern. The same color represents patterns within the same cluster. Clusters shown in red indicate lower quality compared to those highlighted in green. Positive mean values indicate an increase in the mean of code smells, while negative values represent a decrease in the mean of code smells. The mean values represent the overall mean change in each metric, respectively.

Architecture			**Design**			**Testability**		
Pattern	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**
Steady Active	1	0.03	Steady Inactive	1	-0.03	Early Active	1	0.08
Steady Inactive	2	-0.02	Steady Active	2	-0.06	Steady Inactive	1	0.07
Late Active	2	-0.03	Early Active	2	-0.07	Late Active	1	0.01
Early Active	2	-0.03	Late Active	2	-0.12	Steady Active	1	0.00
Implementation			**Test**			**Overall**		
Pattern	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**
Steady Inactive	1	4.16	Early Active	1	0.43	Steady Inactive	1	4.30
Late Active	2	1.80	Late Active	1	0.15	Late Active	2	1.82
Steady Active	2	0.84	Steady Inactive	1	0.12	Steady Active	2	0.86
Early Active	2	0.26	Steady Active	1	0.03	Early Active	2	0.66

As shown in Table 4.3, the *steady inactive* pattern has the highest mean (4.30) for code smell changes, indicating a greater increase in code smells after adoption compared to other release-wise refactoring patterns, leading to reduced code quality.

Steady active shows better performance in reducing coupling and improving cohesion, but it exhibits the worst release-wise refactoring pattern in terms of reducing architectural code smells. This suggests that excessive refactoring at the implementation or design level could potentially reduce the architectural quality of the code, indicating regular maintenance and long-term planning for eliminating architectural code smells. Therefore, it is recommended that developers, if they adopt this pattern, pay attention to the overall structure of the code and avoid making significant changes that might increase architectural code smells while addressing other code smells.

Overall, we observe that the late active refactoring pattern exhibits the best performance in terms of reducing code smells and improving code quality measures by code metrics compared to the others. It minimizes increases in all types of code smells while maintaining high cohesion and low coupling. This suggests that a dedicated refactoring approach might be more effective, where refactoring focuses on improving code quality before each release. Therefore, ***it is recommended that developers allocate an incremental and dedicated time frame for refactoring before each release.***

$RQ_{4.2}$ Summary

The *late active* release-wise refactoring pattern demonstrates the best performance in decreasing coupling, increasing cohesion, and reducing code smells. Furthermore, while the *steady active* release-wise refactoring pattern shows a good performance in reducing coupling and improving cohesion, it demonstrates the poorest performance in reducing architectural code smells. This suggests that excessive refactoring may play an opposite role for architectural code smells.

4.3.3 $RQ_{4.3}$: Does the usage of release-wise refactoring patterns change over time, and how do developers switch from one pattern to another?

Motivation

We identify four primary refactoring patterns related to software releases: *early active*, *late active*, *steady active*, and *steady inactive*. Among these, the *late active* release-wise refactoring pattern is observed the most effective. However, the utilization of these patterns in different stages of development or how to switch from one

pattern to another is not clearly understood, specifically in terms of their implications to code quality. This research question aims to provide deeper insights into the evolution of refactoring release patterns over time, examining how and why developers switch between these patterns and their respective impacts on refactoring quality. This understanding allows practitioners to comprehend existing patterns and evaluate how transitioning between them can enhance code quality through refactoring efforts. We aim to provide a road map that developers can follow when performing refactoring in a CI/CD environment.

Approach

To analyze how the utilization of release-wise refactoring patterns changes over time for a project, we define four stages of development over the lifetime of a project and measure the utilization rate of each pattern throughout these stages. Additionally, we assess the probability of switching from one pattern to another and its effect on code quality. The following presents a breakdown of each step in this process:

Analyzing pattern utilization: To understand and measure the utilization of refactoring release patterns over time, we follow a similar methodology as in previous work [124]. We divide consecutive releases into three time quartiles, each containing an equal number of releases, representing different stages of the project's lifecycle. Therefore, using the three time quartiles, we distribute releases as follows:

- **Early stage:** First 3 releases (1^{st} quartile).
- **Middle stage:** Releases 4 to 6 (1^{st} to 2^{nd} quartile).
- **Late stage:** Releases 7 to 13 (2^{nd} to 3^{rd} quartiles).

- **Last stage:** More than 13 releases (3^{rd} quartile).

This approach allows us to analyze changes in pattern adoption over time by comparing the distribution of each release-wise refactoring pattern across different stages of development. This helps identify the most and least popular patterns and shows how their popularity shifts as software undergoes more releases or matures.

Measuring quality of pattern transitions: To study and analyze the transitions between patterns and their relationship with code quality, we conduct a two-step experiment. First, we calculate the transition probabilities between patterns using Markov chains [141], which model the likelihood of moving from one state (i.e., release-wise refactoring pattern) to another, where each transition depends on the previous state. Second, we use Scott-Knott-ESD [108, 109] to measure and rank code smell changes associated with each transition, grouping them based on the significance of the reduction rate of code smells between the transitions. This two-step approach allows us to identify common shifts in refactoring practices, leading to a deeper understanding of how and why developers adopt and change refactoring patterns throughout the project's lifecycle and their impact on code quality.

Findings

The distribution of the utilization of release-wise refactoring patterns across different stages of development is depicted in Figure 4.5. Furthermore, the results of our two-step approach, which incorporates Scott-Knott-ESD with the probabilities of transition from one approach to another, are shown in Table 4.4.

Developers are more likely to repeat the previous refactoring pattern in later releases, as shown in Table 4.4, particularly in the *steady inactive* (39%)

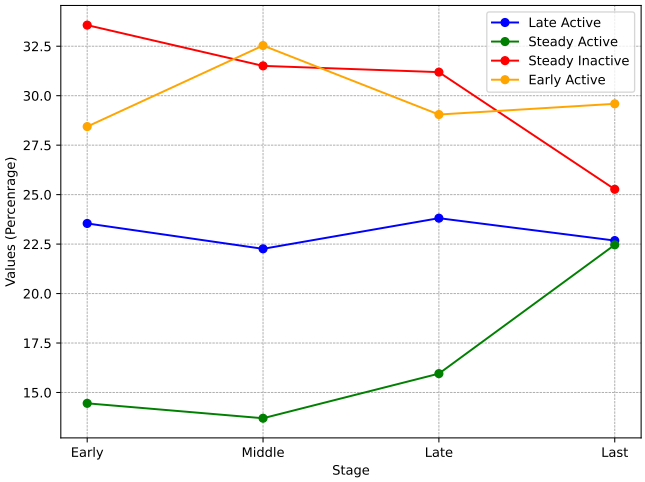


Figure 4.5: The distribution of different refactoring release patterns at various stages of development. Each point represents the percentage of utilization of each pattern on the y-axis, while the x-axis indicates the development stage at which the pattern is utilized.

refactoring pattern, where the refactoring density is the lowest compared to other release-wise refactoring patterns. The *steady inactive* state is the most probable, with a 39% probability of occurring, and shows the least effectiveness in eliminating the total code smells. It ranks worst (Rank 1) in achieving high cohesion (mean 6.72), worst in reducing complexity (mean 2.73), and second worst in lowering coupling (mean 7.02). Therefore, ***it is recommended to avoid staying in the steady inactive state if the goal is to improve quality.***

Even though the likelihood of transitioning from a steady inactive to a steady active pattern is the lowest (9%), the results from the Scott-Knott-ESD show that switching from steady inactive to steady active can effectively eliminate code smells and improve code quality. Specifically, this transition exhibits the second-best improvement in reducing coupling (mean of

2.57), the best improvement in achieving higher cohesion (mean of 2.26), and the best reduction in complexity (mean of 0.97), but a mean increase in architecture smells (mean of 0.01). Additionally, the 9% switch probability rate may indicate less interest or difficulties in shifting from the steady inactive to the *steady active* pattern. ***It is therefore recommended to switch from the steady ease pattern, characterized by minimal refactoring throughout the release, to the steady active pattern, which involves consistent more intensive refactoring throughout the release. However, be aware of potential architectural problems that may arise from the sudden increased refactoring.***

Figure 4.6 illustrates the possible transitions between patterns, highlighting their mean relationship with code smell reduction, reported from the Scott-Knott-ESD test. Safe states, where remaining consistently reduces code smells, are marked in green, while not-safe states, where no transition exists to another state without increasing code smells, are marked in red. As shown, switching from the *late active* to the *early active* pattern positively affects quality, reducing code smells by a mean of 0.81. However, switching back to *late active* results in a decrease in quality, with a mean increase in code smells of 1.29. Therefore, ***it is recommended to avoid deferring refactorings to post-release and instead perform refactorings and cleanings closer to release dates.*** As shown in Figure 4.5, ***the increased utilization of the steady active release-wise refactoring pattern by 8% from the early to the later stages of development suggests more frequent and intensive refactoring efforts compared to earlier stages, where usage decreases by 8.3%.*** This observation may indicate that as projects become mature, they require more maintenance, and the frequency of active refactoring patterns

Table 4.4: Results of the Scott-Knott-ESD test results on the total changes in code smells (per 1,000 lines of code) after transitioning between release-wise refactoring patterns. Patterns within the same cluster (rank) are represented by the same color. Positive mean code smell values indicate an increase in the mean of code smells, while negative code smell values represent a decrease in the mean of code smells (*i.e.,* increase in code quality). Abbreviations used: Arch. = Architecture, Des. = Design, Impl. = Implementation, Cohes. = Cohesion, Coup. = Coupling, Compl. = Complexity.

			Code Smells				**Code Metrics**		
From	**To**	**Rank**	**Total**	**Arch.**	**Des.**	**Impl.**	**Coh.**	**Coup.**	**Compl.**
Steady Inactive	Steady Inactive	1	5.03	-0.05	-0.03	5.1	6.72	7.02	2.73
Early Active	Steady Inactive	2	3.98	0.01	0.12	3.61	5.10	7.24	2.19
Steady Active	Steady Inactive	3	2.80	0.02	0.09	2.46	2.41	2.54	1.07
Steady Active	Late Active	4	2.78	0.01	-0.22	3.11	2.93	3.44	1.24
Early Active	Steady Active	5	2.28	0.02	-0.01	1.92	2.95	3.04	1.27
Early Active	Early Active	6	2.11	-0.08	0.07	1.60	2.95	3.82	1.53
Steady Inactive	Late Active	7	1.73	-0.05	-0.22	2.01	3.67	4.45	1.55
Early Active	Late Active	8	1.29	0.00	0.35	0.59	2.75	3.35	1.31
Steady Inactive	Early Active	9	0.9	0.02	-0.22	0.86	3.35	4.07	1.54
Late Active	Steady Active	10	0.86	-0.01	0.01	0.74	2.29	2.72	1.08
Late Active	Steady Inactive	11	0.85	-0.05	-0.25	0.93	4.27	4.70	1.80
Steady Active	Steady Active	12	0.58	0.06	-0.05	0.60	3.71	3.83	1.69
Steady Active	Early Active	13	0.42	-0.06	0.11	0.19	2.48	2.78	1.03
Steady Inactive	Steady Active	14	-0.11	0.01	-0.08	-0.08	2.26	2.57	0.97
Late Active	Late Active	14	-0.71	-0.09	-0.36	-0.20	2.65	3.32	1.23
Late Active	Early Active	14	-0.81	-0.01	-0.33	-1.09	2.78	3.53	1.30

increases over time. Conversely, the reduced usage of the *steady inactive* release-wise refactoring pattern may be attributed to the growing need for refactoring as the project progresses.

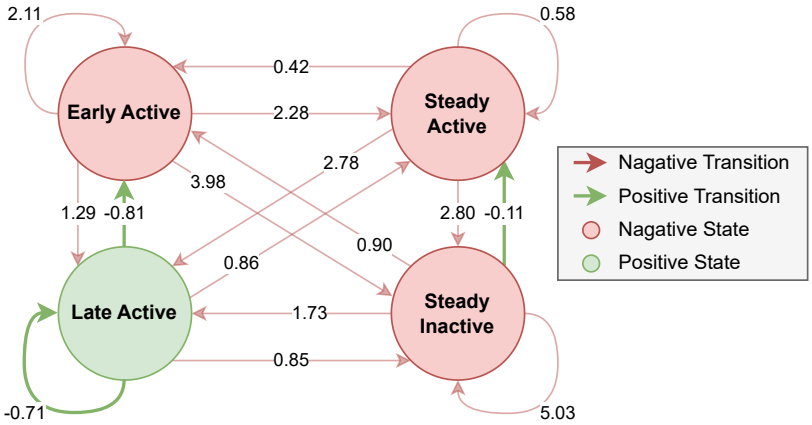


Figure 4.6: Summary of positive transitions (green lines) and negative transitions (red lines). Positive transitions indicate an average reduction in code smells after adopting each pattern, while negative transitions show an increase in code smells. The green circle represents a safe transition that can be adopted repeatedly.

$RQ_{4.3}$ Summary

Developers tend to repeat previously used patterns rather than frequently switching to different ones. However, switching to a different pattern can often lead to a better quality impact. In other words, staying with the same pattern is not the best option in terms of code quality. The usage of the *steady active* pattern increases as projects progress; however, *late active* may be a better alternative, which is the most reliable release-wise refactoring pattern, remains consistent throughout the project lifecycle, and is considered a safe pattern for continuous use. Moreover, postponing refactoring to post-release can temporarily increase quality but may lead to decreased software quality if reverting to the *late active* state. The *steady inactive* release-wise refactoring pattern can be followed by the *steady active* pattern for improved quality.

4.4 Threats to validity

In this section, we examine the potential threats to the validity of our study.

Threats to Construct validity. Indicate the data preparation and feature selection. For generating refactoring time series, we opt to calculate the refactored lines of code divided by the overall code churn. The refactored LOC is calculated from the 99 types of refactoring detected by RMiner. However, if there are additional types of refactoring that RMiner does not detect, we cannot capture them. Therefore, using a more comprehensive tool would reflect a greater number of refactorings in our generated refactoring release time series. For the external features that impact the switching, we measure the features detectable from the commit messages and project repository. However, incorporating more socio-technical information could provide additional insights into the causes of the switching between patterns. Lastly, for measuring refactoring release pattern quality, we use code smells as indicators of refactoring quality. However, incorporating additional metrics, such as more code metrics and attributes related to code quality—such as reliability and bug resolution rates—could provide further insights into the potential benefits of different refactoring release patterns on various aspects of the code.

Threats to Internal validity. Indicate the validity of the methods applied in our study. We choose the Java language domain for our studied projects because Java has the most reliable refactoring tool [10, 21], which detects the highest number of refactoring types. However, if comparable tools are available for other languages, studying projects written in those languages could reveal additional refactoring patterns. For the number of clusters (*i.e.,* refactoring release patterns), we opt for the cluster number with the most distinct centroids while maintaining good separation

of data on the scatter plot. This approach reveals the most dominant refactoring release patterns. However, some minority refactorings could be miscategorized into these clusters. Additionally, our analysis is based on cluster centroids, so there may be some outliers that do not fit neatly into the identified patterns. To determine the quality of release-wise refactoring patterns, we rely on code smells with the addition of code metrics as quality indicators. However, other metrics may also provide valuable insights and could impact the quality assessment differently.

Threats to External validity. Concern about the generalizability of our approach. Our experiments and results are based exclusively on analyzing the 207 open-source projects that follow the standard release naming convention. Consequently, our conclusions may not apply to other projects, particularly those in different domains. Since our analysis is restricted to projects written in Java, the findings may not extend to projects written in other programming languages. Furthermore, our focus on identifying common patterns means that less common and infrequent patterns may be overlooked.

4.5 Conclusions

In this chapter, we analyze 207 popular and active open-source Java projects to identify the common refactoring patterns employed by developers within software release cycles. We first examine refactoring density frequencies and analyze changes in refactoring practices throughout the release cycle, identifying four major refactoring release patterns: *early active*, *late active*, *steady active*, and *steady inactive*. We find that the *late active* release-wise refactoring pattern, which is consistently used across different stages of development, represents the best refactoring practice

to be applied continuously. This pattern is characterized by consistent class, package, and organizational-level refactoring throughout the release, with an increased focus on method and variable-level refactorings as the release date approaches—leading to better code quality post-release. However, the steady active pattern is increasingly adopted by practitioners as projects progress. While the *steady active* pattern, characterized by a high density of refactoring throughout the release, may improve software quality, it can also introduce architectural smells. For future work, we aim to explore automated methods for assessing releases in response to maintenance needs.

Chapter 5

Detecting Refactoring Commits in Python Projects

This chapter describes our proposed stereotype tool for detecting refactoring in Python ML projects. Section 5.1 presents the problems and motivation of our study. Section 5.2 describes the experiment setup of this study. Section 5.3 presents the motivation, approaches, and results of our research questions. Section 5.4 explains the generalizability of our approach to other technical domains. Section 5.5 discusses the threats to the validity of our findings. Finally, we conclude this chapter and present future research directions in Section 5.6.

5.1 Problem and Motivation

Understanding the appropriate timing and implementation of refactoring in software development enables developers to comprehend software evolution, make informative decisions, and enhance their knowledge of code design [19, 20]. However, it is challenging to gain insights into maintenance practices, as developers often provide

insufficient documentation regarding their refactoring activities. Quite often, developers use general terms (*e.g.,* clean-up) to describe the refactoring actions instead of providing specific details [20].

Sophisticated tools (*e.g.,* Rminer [10,21]) are available for other programming languages, such as Java. However, refactoring detection tools for Python are relatively limited. Moreover, existing refactoring detection tools are not specifically designed or tested to identify domain-specific (*e.g.,* ML) refactorings. The existing refactoring detection tools primarily rely on pattern matching using Abstract Syntax Tree (AST) [10, 19, 21] to detect refactoring operations. Consequently, they are unable to identify accessory information in the code, such as logical refactorings that do not follow a specific pattern. These refactorings are comprehended by developers but cannot be identified by pattern-matching in the AST structure changes.

Python has become one of the most popular programming languages, especially for developing Artificial Intelligence (AI) and Machine Learning (ML) libraries, frameworks, and applications [22–25]. The utilization of ML frameworks and libraries has increased in recent years, and many software projects now depend on ML libraries and frameworks [142, 143]. Therefore, ensuring the proper maintainability of the ML libraries and frameworks is essential. Due to the data-driven nature of ML projects, they undergo a different development process, including design and implementation, which is more complex than traditional software development [144–146]. Various refactoring experiences and types may exist in ML-specific contexts, e.g., data modifications like switching dataset types (*e.g.,* from CSV to Parquet) or altering the way of iterating through dataset records.

Compared to other programming languages, Python prioritizes readability [53, 54]

and has a simpler coding structure [54, 55]. For instance, Python is a dynamically typed language, while languages like Java are statically typed [54]. Consequently, identifying class and object relationships through static code analysis in Python is affected. Abstract syntax tree in Python provides less detailed information compared to languages like Java. Moreover, there are no expectations to improve Python AST building tools in the near future [68].

Existing approaches for refactoring detection can be classified into two categories: rule-based [29–32] and keyword-based. However, not all refactoring activities are explicitly mentioned in commit messages. Therefore, keyword-based approaches may fail to detect refactoring commits that lack explicit keyword mentions in their commit messages.

PyRef [19] is the most effective and validated rule-based refactoring detection tool for Python. PyRef leverages pattern-matching within the Abstract Syntax Tree (AST) using a predefined set of rules [19]. PyRef can detect up to 11 types of refactoring operations, covering 11% of the refactoring operations that Rminer can detect in Java code. PyRef achieved a precision of 89.6% and a recall of 76.1%. Additionally, PyRef has not been specifically designed or validated for the ML technical domain for refactoring detection, which may provide incomplete coverage of refactoring operations and increase the likelihood of false negatives in identifying refactoring commits.

In short, the keyword-based and rule-based approaches are often limited by the available keywords and rules in the studied projects. Such approaches may not work well when applied to new projects that use new keywords or unknown rules, or when used in domain-specific projects. As a result, the coverage of these approaches could

decrease in new projects. In contrast, MLRefScanner integrates code, process, and textual features for refactoring detection—a topic not extensively studied but potentially valuable for future research in this area. The objective of our proposed machine learning approach is to extract and learn from the knowledge in existing projects and extend it for refactoring detection, particularly in scenarios where rule-based or keyword-based approaches are not sufficient to recognize new rules or keywords used in the commit messages in new projects.

In this chapter, we introduce MLRefScanner, a purpose-built prototype tool designed specifically to identify refactoring commits in the history of ML Python projects. Leveraging machine learning techniques and algorithms, MLRefScanner aims to improve the coverage of identifying refactoring operations in Python code and enhance the capabilities of detecting ML-specific refactorings. MLRefScanner allows practitioners and researchers to track and analyze refactoring activities with high precision and recall in the software development history of ML Python libraries and frameworks, enabling a deeper understanding of the scope and evolution of the ML codebase.

To evaluate the effectiveness of MLRefScanner, we conduct empirical studies on 199 ML libraries and frameworks. In this study, we aim to answer the following research questions:

***RQ*$_{5.1}$**: What is the performance of our approach for identifying refactoring commits in ML Python projects?

> To identify refactoring commits in the history of ML Python libraries and frameworks, we analyze their commit history and extract multiple dimensions of representative features (*e.g.,* commit message-related features) to train MLRefScanner with different classifiers (*e.g.,* Random Forest).

Our findings demonstrate that LightGBM is the best-performing classifier for detecting refactoring commits, enabling MLRefScanner to achieve an overall precision of 94%, a recall of 82%, and an AUC of 89%. MLRefScanner can identify refactoring commits with ML-specific and additional refactoring types compared to state-of-the-art tools.

***RQ*$_{5.2}$**: What are the main features for explaining refactoring commits?

To gain deeper insights into the nature of refactoring commits for practitioners and increase the transparency of the proposed approach, we apply a features evaluation approach using the number of decision splits and the LIME interpreter [147], which helps us identify the most important features of refactoring commits. As a result, the top 10 features that indicate a refactoring commit include: (1) textual features: *update, remove, refactor, improve, and move*; (2) process features: *author's refactoring contribution ratio, lines deleted, lines added, code entropy*; and (3) code features: *lines of declarative code*.

***RQ*$_{5.3}$**: Can we leverage commit information to complement existing keyword-based and rule-based approaches?

We compare MLRefScanner with existing keyword and rule-based approaches in identifying refactoring commits, demonstrating its superiority. Our approach outperforms the existing best approach (*i.e.,* PyRef) achieving a 22% increase in AUC. Additionally, we explore the potential benefits of utilizing ensemble learning to complement PyRef with MLRefScanner. The ensemble learning approach achieves an overall precision

of 95% and a recall of 99%, significantly improving the low recall of the state-of-the-art approaches.

The replication package for this chapter is available online[1].

5.2 Experiment Setup

This section presents the setup of our study, including our data collection and data analysis approaches.

5.2.1 Overview of Our Approach

We propose an ML-based approach to detect refactoring commits in ML Python projects. An overview of our study is depicted in Figure 5.1. We train MLRefScanner in the machine-learning domain using 199 ML libraries and frameworks. Leveraging the commit history on the subject projects, we create a curated dataset by labeling each commit as a refactoring commit or not, based on the commit messages and refactoring analysis results obtained from the PyRef tool [19]. We manually validate our labeling approach through a validation process involving two individuals to assess its performance. Subsequently, we extract textual, process, and code features to capture the characteristics of each commit. We then partition the data into training sets and testing sets. During the training phase, we preprocess the data by applying undersampling to address issues related to an unbalanced dataset. Additionally, we conduct hyperparameter tuning to optimize the training settings of our classifiers. We interpret the trained model and extract the most important features to provide deeper insights into the characteristics of refactoring and non-refactoring commits.

[1] https://github.com/Shayan-Noei-PhD-Replication-Packages/Chapter-5-Detection

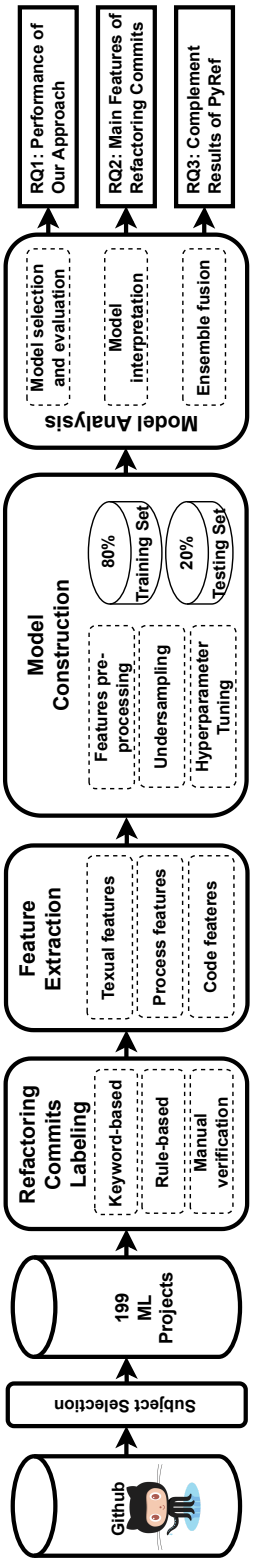


Figure 5.1: Overview of the approach.

Finally, we ensemble the proposed model with the state-of-the-art refactoring detection tool (*i.e.,* PyRef) and assess whether we can enhance the refactoring detection performance by considering a variety of features in each commit.

5.2.2 Subject Selection

We select machine learning projects to train and test MLRefScanner in the ML technical domain. Inspired by previous work [148], we use the advanced search on GitHub[2] to select repositories of machine learning libraries, frameworks, and applications primarily written in Python by specifying keywords, including *deep learning, reinforcement learning,* and *machine learning*. As a result, we identify 161,662 projects. It is time-consuming to extract refactoring commits. For example, it takes PyRef[3] approximately one day to process each project because it involves parsing the commit code change history and constructing an AST for each commit. We plot the distributions of project size (lines of code), stars, and forks of the repositories and adopt a systematic process [148] to select the projects. The selection criteria help us select mature and active projects that have demonstrated sustained growth and community engagement. We remove the projects that have a size smaller than 23,148 KB, fewer than 27 stars, or fewer than 18 forks as such projects are below the 3rd quartile of the distribution; have had no pushes in the year leading up to the data collection date (from 2021/04/01 to 2022/04/01); have no issues; have no downloads; are archived; are disabled; or have fewer than 1,000 commits. As a result, we obtain 217 repositories. Furthermore, we perform a manual validation to remove 18 irrelevant (*i.e.,* non-ML repositories or repositories that contain only documentation).

[2] https://docs.github.com/en/rest/search?apiVersion=2022-11-28
[3] We use PyRef in our data curation, as described in Section 5.2.3.

Finally, 199 repositories are selected as the basis (*i.e.,* training and testing) for our experimental analysis.

5.2.3 Labeling Refactoring Commits

To label commits as either refactoring or non-refactoring in the selected ML projects, we employ two state-of-the-art automated approaches: (1) keyword-based labeling and (2) rule-based labeling with PyRef. If a commit is identified as a refactoring by either approach, we label it as a refactoring commit. During the labeling process, we classify 155,780 commits as refactoring and 343,187 as non-refactoring commits. We acknowledge that the automated labeling approach may lead to false positives or missing potential refactoring commits. Thus, we perform manual validation on a statistically significant sample size to verify the precision and recall of the automated labeling results. The primary purpose of our manual labeling is to illustrate the overall correctness of the labeled dataset used for training and testing the machine learning models, rather than to account for every minor change or exception in the detection of the refactoring-related commits. In particular, this automatic labeling approach is designed to build MLRefscanner, not to cover all possible cases comprehensively. Moreover, the automatic labeling approach can have decreased performance when it is applied to new projects or domain-specific projects.

Keyword-based labeling

We compile a list of keywords that indicate a refactoring commit (*e.g., modify, enhance,* or *clean-up*) used in previous studies [29–31, 69]. These keywords are typically associated with self-admitted refactoring operations, meaning that developers

explicitly mention in the commit messages that refactoring is performed. To create a starting point, we manually investigate each keyword using a small set of 10 sample commits that contain the keyword and assess if they consistently represent refactoring commits. This validation process helps us minimize false positive keywords and the manual validation cost. We exclude keywords that do not consistently relate to refactoring operations. For instance, we find that certain keywords, such as *fix* might not always represent refactoring operations. The final set of chosen keywords used to label a refactoring commit includes: *move, refactor, remove, rename, split, clean, improve, unused, cleanup, simplify, restruct, inline, parameterize, consolidate, encapsulate,* and *update*. We then use the keywords to match the commit messages of all the commits of the studied projects: a commit is labeled as a refactoring commit if it contains at least one of the keywords. We use an exact match search, which sets a commit as a refactoring commit if any part of the commit message contains the specified keywords. In the labeling process, we also classify commits that do not alter any executable code (*e.g.,* only changes in the ReadMe file) as non-refactoring commits to ensure that the labeled refactoring commits involve changes to the executable code.

Rule-based labeling

Not all refactoring operations are self-admitted; some may be performed without explicit mention in the commit message. As PyRef can achieve a precision of 89.6% and a recall of 76.1% in detecting refactoring operations in ML Python projects, we utilize PyRef [19] to account for refactoring activities not mentioned in the commit message. If PyRef identifies any refactoring operations, we label the commit as a refactoring commit even if there are no specific refactoring keywords in the

Table 5.1: The set of features utilized in the study. Textual features include vocabulary and features extracted from commit messages; process features encompass details processed from commits; and code features comprise code metrics extracted from the source code before and after each commit code changes.

Feature Type	**Name of Features**
Textual	TF Vectorized Commit Messages (Vocabulary), Words Count, Sentences Count, Readability
Process	Lines Added, Lines Deleted, Lines Changed, Number of Files, Code Entropy, Refactoring Contribution Ratio, Has Executable File
Code	Avg Count Line, Avg Count Line Blank, Avg Count Line Code, Avg Count Line Comment, Avg Cyclomatic, Count Class Base, Count Class Coupled, Count Class Coupled Modified, Count Class Derived, Count Decl Class, Count Decl Executable-Unit, Count Decl File, Count Decl Function, Count Decl Instance Method, Count Decl Instance Variable, Count Decl Method, Count Decl Method All, Count Line, Count Line Blank, Count Line Code, Count Line Code Decl, Count Line Code Exe, Count Line Comment, Count Stmt, Count Stmt Decl, Count Stmt Exe, Cyclomatic, Max Cyclomatic, Max Inheritance Tree, Max Nesting, Ratio Comment To Code, Sum Cyclomatic

commit message. The results obtained from PyRef complement the keyword-based labeling results. As PyRef can detect 11 types of refactoring operations including *rename method, add parameter, remove parameter, change/rename parameter, extract method, inline method, move method, pull up method,* and *push down method*. PyRef is used to label our dataset in identifying commits that involve these refactoring operations.

Manual validation

To assess the accuracy and reliability of our labeling approach, we engage the author of the thesis and a third-year undergraduate student in software engineering to conduct manual validation. In cases where there is disagreement, a third-year PhD student in software engineering labels the commit, and consensus is reached among all validators on a final label. Before splitting our dataset into training and testing

sets, we randomly select 383 commits from the dataset to ensure that they are not included in either the testing or training sets. This selection is made to maintain a 95% confidence level with a 5% margin of error. Both validators independently assess the commit messages and code changes of the commits and label them based on whether they observe a refactoring operation or not. If they observe one, they label it as a refactoring commit. Cohen's kappa agreement level [82] is found to be 0.64 after the initial labels, indicating a moderate level of agreement [149] between the validators. The validators discuss the initial labels they provided on each commit to reach a consensus on a final label. The final manually validated set includes 76 refactoring commits and 307 non-refactoring commits. By comparing the labels of 383 commits agreed upon by the validators and the labels generated by our automatic labeling approach for the same 383 commits, we achieve a precision of 91% and a recall of 90% using our automatic labeling approach (*i.e.,* using the agreed-upon manual labeling results as the ground truth). In detail, the automated labeling process identifies 5 false-positive and 5 false-negative labels. We observe that the main reason for the false negatives is that the PyRef is incapable of identifying all refactoring types, thus failing to detect refactoring operations in certain commits and reporting them as non-refactoring. Moreover, the reason for the false positives is the misinterpretation of certain keywords, such as *improve*, which do not necessarily describe refactoring commits. For instance, we observe a commit with the message "fix(helper): improve collision in random_port" containing the *improve* keyword, but it is not related to refactoring. Consequently, we use our manually labeled results as the ground truth for further analysis and evaluation of our approach.

5.2.4 Feature Extraction

To identify refactoring-related commits, we consider three dimensions of features that can be extracted from commit logs to effectively detect refactoring commits, including (1) textual features, (2) process features, and (3) code features, as listed in Table 5.1. Textual features can reveal refactoring commits through commit messages and developer communications, by explicitly mentioning refactoring operations. For example, a commit message may explicitly mention a refactoring-related keyword, such as *refactoring*. Process features are used to measure characteristics of commits that might indicate the presence of refactorings. For example, code entropy may show how diverse the code changes are among different files. Higher entropy potentially indicates the refactoring operations related to code movement across files. Additionally, code features describe the actual code changes and modifications quantified using code metrics, which can reveal underlying patterns of refactoring.

Textual features

The textual features are listed in Table 5.1. We first clean up the commit messages to identify the vocabulary in the commit messages. We convert all messages to lowercase to ensure that the capitalization of words does not affect the classification. To address typos and mistakes in commit messages, we utilize the Enchant library [150] to perform spelling corrections and unify word usage, regardless of whether they are mistakenly written or not. We remove URLs, HTML tags, punctuations, emojis, numerical digits from the messages, and stop words (*e.g.,* an, with, or but). Moreover, we stem the commit sentences to their root [151] form to ensure that variations of words (*e.g.,* fixing and fix) are treated as the same. We choose stemming to avoid

duplication and improve the generalization of the features. For instance, when using Lemmatization, *refactor, refactoring,* and *refactored* are considered separate features. In contrast, with stemming, we handle all of the variations of the word with a single vector as *refactor*, which prevents our model from being overwhelmed by the explosion of the contextual meanings of unique words in the commits. Term frequency (TF) [152] in commit messages is used to retain the ability to interpret and understand the contributions of individual words in the classification process, which assigns a value of 0 or 1 based on the presence or absence of a word. We opt for TF instead of TF-IDF (term frequency-inverse document frequency) because TF-IDF tends to assign lower weights to the most repeated words. However, in our case, certain words like *refactor* may appear frequently and should be weighted more. Additionally, we choose not to use vector transformation algorithms like Bert [153] or word2vec [154] for commit messages as they are not reversible and interpretable. Using such algorithms would prevent us from investigating the root cause and important features that distinguish a commit based on whether it involves refactoring or not. In the vectorization setup, we use 6-grams (*i.e.,* a maximum of 6 consecutive words as one term) to extract sufficient information from the commit messages, thereby improving our ability to identify potential refactoring activities. The choice of 6-grams is based on a previous study on self-admitted refactorings [29], which identify the maximum number of words in a sentence that can potentially reveal a refactoring commit as 6. In addition, we calculate textual features, such as word count and sentence count for each commit message. To assess the readability of these commit messages, we employ the Flesch reading ease metric [155], which indicates the level of comprehension difficulty in the commit messages.

Process features

We collect a set of process features that capture various characteristics of commits, as outlined in Table 5.1. We analyze the modified code blocks and files within each commit using the *git log* information on the cloned repositories to extract features including *lines added, lines deleted, lines changed, has executable file,* and *number of files*. This information helps us better describe the scope of code changes in each commit. The refactoring contribution ratio (RCR) provides information on the experience of each code author performing refactoring-related commits within each project. To calculate the RCR, we collect all labeled commits of each code author before each commit (as described in Section 5.2.3) and use the formula below:

$$RCR(i) = \frac{Refactoring\ commits\ of\ the\ author\ (i)}{All\ commits\ of\ the\ author\ (i)} \tag{5.1}$$

To gain insights into the complexity and structural changes within each commit, we calculate the code changes entropy using Shannon's entropy [156], which has been utilized for measuring the complexity of code changes [157]. For each file within a commit with executable code, we calculate the relative code churn of that file by dividing the code churn of that file by the total code churn of the commit, denoted as $P(x_i)$. Then, we use Shannon's entropy formula to calculate the code change entropy within each commit. The formula for Shannon's entropy is as follows:

$$H(x) = -\sum_{i} P(x_i) log P(x_i) \tag{5.2}$$

Code Features

To capture information about code changes within each commit, such as code complexity, we calculate code features related to the code changes of each commit. We extract the modified files associated with each commit. Subsequently, we calculate 32 sets of code features in class, function, and code levels, as outlined in Table 5.1, before and after each commit. We use the Understand[4] tool to calculate code features [158–160] for the current and previous state of each commit, then we compare the current and previous state of changes and calculate the difference of the code features (*i.e.*, reduction or increase) for each code feature.

Feature pre-processing

The existence of highly correlated and redundant features could affect the stability of the model and the interpretation of features [94, 161]. Therefore, on the training set (described in Section 5.2.5), we conduct correlation analysis and redundancy analysis of the process and code features. We don't perform correlation and redundancy analysis on the textual features as it may lead to some loss of information and nuances during model interpretation. Each textual feature can provide critical insights into why the model has reached a particular decision, and removing correlated textual features can present challenges when interpreting the decision-making process of the model.

- **Correlation analysis:** Our features do not follow a normal distribution, therefore we utilize Spearman's correlation coefficient to determine the correlation between the computed features. A coefficient > 0.7 indicates a strong correlation [94, 162].

[4] https://scitools.com/

For strongly correlated features, we choose to retain one feature in our model while removing the other. The results of the correlation analysis show that *Number of Files, CountClassBase, CountDeclExecutableUnit, CountLine, CountLineCode, CountLineCodeExe, CountStmt, CountStmtDecl,* and *CountStmtExe* are highly correlated with other features. Therefore, we exclude them from our features.

- **Redundancy analysis:** R-squared is a metric used to indicate how much of the variance in a dependent variable can be explained by independent variables [95]. To identify redundant features that can be estimated from other features, we use one feature as a dependant and other features as independent features [163] setting a cut-off value of 0.9 [164] for the R-squared. Following the removal of highly correlated features, we conduct the redundancy analysis, yet no redundant features are found.

- **Removing low-variance features:** We exclude features with a variance threshold lower than 0.001 in the training set. This threshold implies that the feature value is identical in 99.9% of the data, indicating a minor impact on the data. This step becomes necessary because our initial textual features include 3,433,690 features. By excluding low-variance features, we reduce the feature size to 651, thus focusing on informative features.

Finally, we use min-max normalization [165] on the process and code features to fit them into the same range as TF vectorized textual features, ensuring that the features are rescaled to the range [0, 1]. For instance, the number of file changes within each commit could vary from [0, ∞]. By utilizing min-max normalization, we rescale it to the range [0, 1].

5.2.5 Model Construction

To build our ML-based approach, we discuss the model construction steps in the following subsections.

Data splitting

We randomly split our dataset into a 20% testing set and an 80% training set. To avoid bias, we leave the testing set untouched throughout the entire model-building process, including features pre-processing, undersampling, and hyperparameter tuning.

Under-sampling of training data

Our training dataset contains a total of 155,780 refactoring commits and 343,187 non-refactoring commits, resulting in an imbalanced dataset. Class imbalance in our dataset has the potential to skew our classification model towards the majority class (*i.e.,* non-refactoring commits), leading to reduced accuracy [166]. As the size of the dataset is large (*i.e.,* 498,975 records with 651 features), oversampling would significantly increase the computational costs. Moreover, undersampling has been shown to be a better approach compared to oversampling [167,168]. Hence, we choose to undersample our training dataset. Random undersampling can lead to data loss during the random selection of majority class samples [169]. Therefore, we perform three different NearMiss undersampling approaches [170] and evaluate our models on each, seeking the best-performed undersampling approach. We apply the following undersampling approach to our dataset:

- NearMiss-1: Choose negative (*i.e.,* non-refactoring) samples with an average Euclidean distance [170, 171] closest to the sum of the three closest positive (*i.e.,* refactoring) samples.
- NearMiss-2: Select negative samples that are nearest to the sum of all closest positive examples.
- NearMiss-3: For each positive sample, select the negative examples that are closest.

Therefore, as discussed in Section 5.3.1, we select 306,197 commits as our final balanced training dataset after undersampling.

Model Training

To train our model, we use eleven different classifiers belonging to three different categories. (1) Linear classifiers: Naive Bayes, Support Vector Machine (SVM); (2) Nonlinear classifiers: Decision Tree, k-Nearest Neighbor, Random Forest, Multi-Layer Perceptron (MLP), and CatBoost; and (3) Ensemble Learning (EL): Adaptive Boosting (AdaBoost), Gradient Boosting Machine (GBM), XGBoost, and LightGBM [172, 173].

Hyperparameter Tuning

To enhance the accuracy of the selected models, we perform hyperparameter tuning on the training set. As discussed in Section 5.2.5, our training dataset consists of 306,197 records after under-sampling and includes the extracted features listed in Table 5.1. Training the models can be time-consuming, with an average training time of around 20 minutes per model. Therefore, we utilize random search to explore the

hyperparameter space and find the best configuration efficiently. Random search, an efficient and effective hyperparameter tuning approach, allows us to sample combinations efficiently and often achieves better results in less time compared to other approaches, such as grid search [174].

5.3 Results

In this section, we provide the motivation, approach, and findings for each of our research questions.

5.3.1 $RQ_{5.1}$: What is the performance of our approach for identifying refactoring commits in ML Python projects?

Motivation

Refactoring is applied in practices to increase software maintainability. It is essential to track previous refactoring activities by analyzing commit information so that developers can gain a better understanding of the state of code. This includes monitoring the history and timing of refactoring applied in the history of code, identifying vulnerable code segments that have undergone frequent refactoring operations, and estimating future maintenance needs [9, 175, 176]. The state-of-the-art refactoring detection tools for Python code are relatively limited. For instance, PyRef can detect 11 types of refactoring in Python code. Moreover, existing tools utilize AST pattern matching to detect refactoring in general repositories, however, the existing tools have not been specifically designed and tested for particular domains, such as machine learning. With the rapid growth in the number and usage of ML libraries and frameworks, predominantly written in Python, our approach, MLRefScanner is

developed to detect commits with more comprehensive refactoring operations within the ML libraries and frameworks written in Python.

Approach

As explained in Section 5.2, we train and evaluate our refactoring commit detection models using our dataset, which contains 199 ML projects. We assess the overall performance of our model based on precision, recall, AUC, and F1 scores. Precision defines the ratio of true positives to the total positive predictions. Recall measures the ratio of true positives to the actual positive instances. F1 score calculates the harmonic mean of precision and recall. AUC represents the ability of the model to distinguish between classes based on probability [177].

Model Evaluation. We evaluate the performance of our model in the following scenarios:

- **Mixed Projects:** We train our refactoring commit detection models using 80% of the commits of the selected projects, and then we assess the performance of our model on the testing set which consists of the remaining 20% of all commits in our dataset from different projects altogether.

- **Cross Projects:** Quite often, some projects may not have a sufficient number of historical refactoring commits to train a model. Therefore, we train our model using all but one (n-1 projects) and assess the performance of our models in the remaining project as testing, where the model is tested on the project excluded from the training dataset. We repeat the leaving-one-out approach for all projects.

- **Within Projects:** To maintain consistent results across different projects and prevent large projects from dominating our dataset, we employ a within-projects setting and test our models separately for each project. We train our model using 80% of the commits randomly sampled from each project and evaluate its performance on the remaining 20% of commit messages from the same project. Specifically, we train one model for each project and test it on the same project.

- **Ground Truth:** To ensure the reliability of our model's performance, we test the final model, which is trained on 80% of all commits from all projects, on the ground truth dataset as described in Section 5.2.3.

Manual Analysis of the Identified Refactoring Types in the mixed-projects Setting. To gain deeper insights into the capability of MLRefScanner in detecting ML-specific refactoring operations, employing a 95% confidence level with a 5% margin of error, we select a set of 383 commits identified as refactoring by MLRefScanner. Then, we ask the author of the thesis and a fourth-year undergraduate computer science student to conduct a thematic analysis to inspect the refactoring patterns identified by the tool in the chosen refactoring commits. Subsequently, they engage in discussions to compare the patterns they have identified and agree on final refactoring patterns in each commit. During the manual validation, the validators consider a portion of code as a refactoring if it does not alter the external behavior of the code [5]. In manual thematic analysis, Cohen's kappa agreement [82] level in commits involving ML-specific or general refactoring activities is determined to be 0.82 between validators, indicating a perfect level of agreement [149].

Table 5.2: The evaluation scores of different classifiers.

Model	**Precision**	**Recall**	**AUC**	**F1**
LightGBM	0.94	0.82	0.89	0.87
CatBoost	0.92	0.83	0.9	0.87
GBMClassifier	0.90	0.83	0.90	0.87
AdaBoost	0.91	0.82	0.89	0.86
XGBoost	0.96	0.76	0.88	0.85
SVM	0.93	0.77	0.87	0.84
MLPClassifier	0.93	0.77	0.87	0.84
Random Forest	0.84	0.78	0.86	0.81
K-Neighbors	0.88	0.74	0.85	0.81
Decision Tree	0.79	0.81	0.86	0.80
Complement NB	0.77	0.72	0.81	0.74

Findings

Best Undersampling Approach: *NearMiss-3* undersampling provides the highest AUC (87%) and F1 (0.84%) scores in the mixed-projects setting. Figure 5.2 presents the evaluation scores for 11 different classifiers, *NearMiss-3* undersampling achieves the highest median precision (91%) and a median recall of 78%. On the other hand, *NearMiss-1* has a higher median recall (94%), but it experiences a significantly lower precision (56%). Additionally, we manually inspect the classification scores in all the classification models and verify the superiority of *NearMiss-3* for all classifiers on our dataset. This can be attributed to the complexity of the decision boundary, as NearMiss-3 selects non-refactoring samples that are closest to refactoring samples, a technique that has also been demonstrated as effective in prior studies [170, 178, 179]. Consequently, we choose *NearMiss-3* as the undersampling technique.

Best Classification Model: The LightGBM classifier exhibits the best

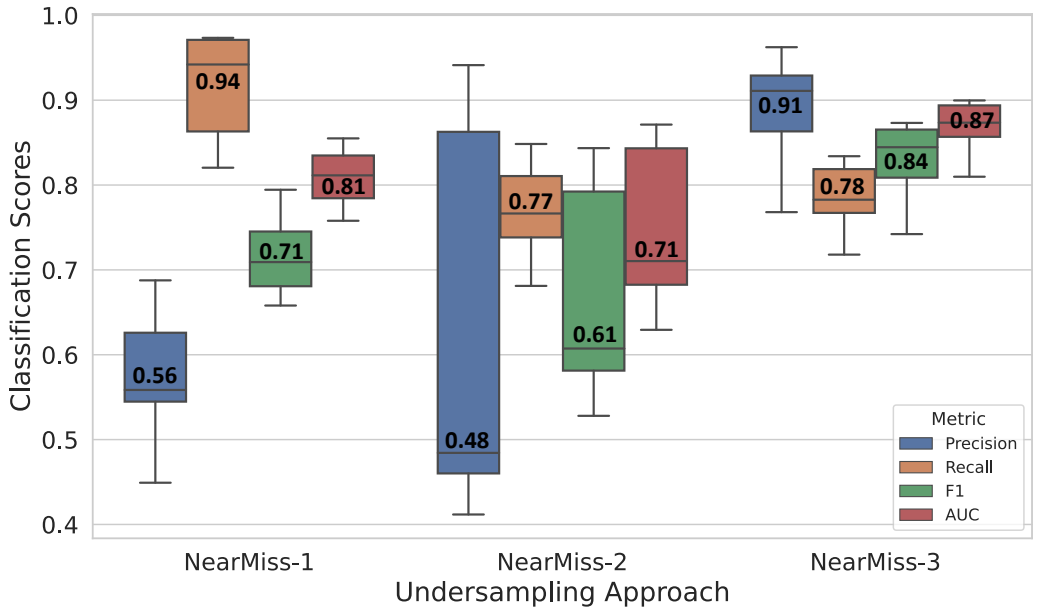


Figure 5.2: Compare the Results of Different Undersampling Approaches

performance in refactoring commit identification, achieving an overall precision of 94%, recall of 82%, and AUC of 89% on the mixed-projects setting. Table 5.2 presents the details of the scores obtained for different classifiers using NearMiss-3 undersampling. Our findings demonstrate that both LightGBM and CatBoost classifiers outperform other approaches, with CatBoost achieving a precision of 92% and a recall of 83% in the mixed-projects setting. Given our objective to maximize precision (*i.e.,* minimize false positives), we select LightGBM as the nominated model and the best-performing classifier for the remainder of our study.

Our approach can not only accurately detect refactoring commits when trained and tested in the same projects (i.e., within-projects), but also in projects that have not been seen before, even if they may exhibit different characteristics (i.e., cross-projects). In within-projects validation, we achieve a median precision of 94%, a median recall of 82%, and a median AUC of 87%. In the

cross-projects validation, we achieve a median precision of 86%, a median recall of 80%, and a median AUC of 90%. The results of the within-projects and cross-projects validations are shown in Figure 5.3. Furthermore, we achieve a precision of 90%, a recall of 81%, and an AUC of 90% on the ground truth dataset.

MLRefScanner can detect commits with ML-specific or general refactoring operations that cannot be detected by state-of-the-art refactoring detection tools. MLRefScanner can detect commits involving both general refactoring operations at the class, method, and variable levels, as well as ML-specific refactoring operations related to data handling, optimization, and ML pipelines. Through manual thematic analysis, we identify a set of ML-specific and general refactoring patterns in the commits detected by MLRefScanner that cannot be identified using state-of-the-art AST pattern-matching refactoring detection tools. ML-specific refactoring patterns are typically related to dataset handling, data reading/writing operations, mathematical computations, and ML model parameter adjustments. On the other hand, general refactorings are broader refactoring operations that do not adhere to a regular pattern; instead, they mainly rely on domain knowledge. For instance, code optimization involves replacing a block of code with a simpler and more understandable alternative. The explanation and the frequency of application for the extended refactoring types are provided in Table 5.3.

MLRefScanner is trained on Python projects but is not limited to Python source files and code. Through manual thematic analysis, we observe that 22 out of 384 commits with source files other than Python have been detected as refactoring commits. This is because MLRefScanner is trained not only using code features but

also a set of process and textual features that are independent of programming languages. Our refactoring detection model uses 651 different features (*i.e.,* keywords, code features, and process features) to decide on a commit. It learns from the automatically labeled dataset to identify refactorings in different projects that don't necessarily follow the rules or keywords used in the labeling process. For example, a commit with the message "add base class for TensorBoard handlers and refine code formatting" is labeled as a non-refactoring commit based on our automatic labeling approach. However, our classifier identifies this as a refactoring commit. This is because our classifier considers not only the entire vocabulary in the commit message (*e.g., add* and *format*) but also incorporates code features and other process features into its decision-making process. Therefore, we assume that MLRefScanner would be more generalizable to unlabelled data than directly using the labeling approach.

To verify this assumption and assess the generalizability of our approach, we manually evaluate 8% of the testing set where the labeled results by the automatic labeling approach differ from our classification approach. The author of the thesis and another graduate student in software engineering independently validate 30% of 382 randomly selected commits, following the same approach as Section 5.2.3, for refactoring, achieving 83% precision and 81% recall, with a Cohen's kappa of 0.90. Due to the excellent agreement and the limited resource available for manual evaluation of the rest of the sampled dataset, the author of the thesis continues to label the remaining commits. For the 382 sampled commits, MLRefScanner achieves 82% precision and 80% recall, while the automatic labeling approach has only 18% precision and 37% recall, confirming the validity of our assumption.

Table 5.3: The list of refactoring types that can be identified by MLRefScanner but cannot be identified by state-of-the-art pattern-matching detection tools.

	Type	Description	Frequency
ML-Specific	Data Handling Optimization	Optimizing data reading/writing from/to the dataset.	14.4%
	Model Initialization Refinement	Modify the hyper-parameter initializations in the models.	04.8%
	Resource Allocation Optimization	Adjusting hardware usage parameters.	04.8%
	Data Path Management	Managing dataset/plots/models storage paths.	03.8%
	Data Type Clarification	Converting data types from one to another (e.g., from NumPy to DataFrame).	02.9%
	Data Presentation Enhancement	Refactoring data visualization code.	02.9%
	Mathematical Operation Refactoring	Changing mathematical calculation to a simpler form.	00.7%
General	Code Cleanup	Removing unused code or files.	16.7%
	Code Simplifications	Replacing code with more simplified alternatives.	13.5%
	Import/Export Optimization	Removing or relocating unused dependencies.	10.9%
	Logging Enhancement	Improving log messages for user understanding.	10.5%
	Requirement/Configuration Update	Updating program requirements or configurations.	08.4%
	Condition Simplification	Simplifying complex conditional statements.	03.0%
	File/Dependency Path Refinement	Optimizing/updating dependency/file path.	02.5%

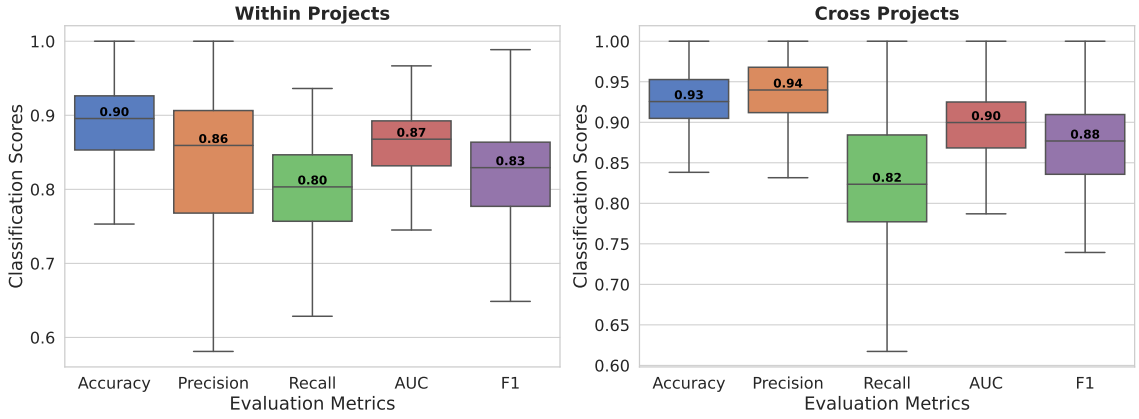


Figure 5.3: Compare the Performance of Our Approach Using Cross and within-projects Validations.

$RQ_{5.1}$: Summary

MLRefScanner can accurately detect refactoring commits within the ML technical domain (*i.e.,* ML projects) in mixed-projects, within-projects, cross-projects, and ground truth settings. MLRefScanner achieves 94% precision, 82% recall, and 89% AUC on the testing set of the selected ML projects in the mixed-projects setting. MLRefScanner is capable of detecting ML-specific (*e.g.,* data processing optimization) and a set of new refactoring types (*e.g.,* Code Cleanup) that are dismissed by state-of-the-art tools relying on AST pattern matching algorithms.

5.3.2 $RQ_{5.2}$: What are the main features for explaining the refactoring commits?

Motivation

In the first research question, we propose a classifier capable of identifying refactoring commits in the ML technical domain. It is important to explain the outcomes from the classifier decisions on detecting refactoring commits. By understanding key

features, we can provide practitioners, especially tool makers, and researchers, with insights into the characteristics of refactoring commits.

Approach

Importance of each feature type. We assess the impact of each feature type (*i.e.,* textual, process, and code) listed in Table 5.1 on the decision-making process of our best-performing model (*i.e.,* LightGBM). To achieve this, we train separate LightGBM models using textual, process, and code data individually on the ML training set and measure the performance of each model on the ML testing set separately. This enables us to (1) indicate the importance of each feature type and (2) demonstrate how each feature type contributes to the identification of refactoring commits.

Individual feature importance. We select the best-performing trained classifier (*i.e.,* LightGBM) to compute the importance of each feature by assessing its frequency and significance in splitting the decision trees during the training process. However, the number of splits in the decision tree doesn't interpret how each feature contributes more towards predicting specific prediction (*i.e.,* refactoring or non-refactoring commit) and the model remains a black box. Therefore, we leverage the local interpretable model-agnostic explanations (LIME) technique [147].

LIME enables us to explain the features of individual instances in our dataset and assigns weights to each feature, indicating their impact on the decision-making process of the model. Using LIME, we gain insights into the model's behavior at the instance level and can better understand how each feature affects the predictions of our model. This approach enhances the transparency and interpretability of the decision-making process of our model.

Explaining feature contributions with LIME. We first use the classifier's splitting score, and then we apply LIME to extract the weight of each feature for each instance in our testing set. A negative weight reported for a feature indicates that the presence or increase in that feature contributes to the decision-making process toward the negative class *(i.e.,* refactoring), while a positive weight indicates that the increase or presence of that feature is associated with the decision-making process favoring the toward class (*i.e.,* non-refactoring). Lastly, a neutral weight implies that the feature does not independently influence the decision-making process but rather relies on other features for classification. We train LIME on our training set and then apply it to every instance in the testing set. We use the median weights of features, and we assign a positive, negative, or neutral weight to each feature, indicating the extent to which a feature is generally associated with refactoring commits.

Findings

Overall, textual features play the most important role in identifying refactoring commits. However, the addition of process and code features to textual features results in an improvement of 7% in precision, 11% in recall, and 7% in AUC. The results of inspecting the impact of different types of features (*i.e.,* process, code, and textual) on our model are shown in Table 5.4. The significant improvement in recall brought by the process and code features enables the identification of refactoring commits that may not be explicitly mentioned in the commit messages.

The appearance of certain vocabulary keywords in a commit message indicates the existence of refactoring in that commit. The most influential

Table 5.4: The results of the trained classifier on each metric type and overall.

Feature Type	**Precision**	**Recall**	**AUC**	**F1**
Textual	0.87	0.71	0.83	0.78
Process	0.52	0.73	0.71	0.60
Code	0.53	0.41	0.62	0.46
Overall	0.94	0.82	0.89	0.87

keywords for classifying a commit as refactoring include *update, remove, refactor, improve, move, rename, clean, cleanup, add, simplify, split, unused, restructure, option, name, parameter,* and *argument*. Specifically, some textual features may indicate a general refactoring operation in a commit (*e.g., refactor*), while others may be more specific, such as *rename*, which indicates the renaming of a variable, method, or class at some level of the code.

Past refactoring contributions of the developers, code complexity features, and code changes are the most common indicators of refactoring. More specifically, increased values of the process and code features such as *refactoring contribution, lines deleted, lines added, code entropy, CountLineCodeDecl, SumCyclomatic, AvgCountLineCode, MaxCyclomatic, CountLineComment,* and *CountLineBlank* play a significant role in influencing the decision of our model towards classifying a commit as refactoring.

The presence of certain vocabulary keywords in a commit message indicates the absence of refactoring in that commit. Keywords such as *readme, doc,* and *fix* are identified as characteristics that lead the model to make a non-refactoring decision. Furthermore, other features, including *Executable File*, *CountDeclFunction*, *Lines Changed*, *Readability*, *words count*, *RatioCommentToCode*, and *ad* vocabulary

Table 5.5: The table presents the most important features in our classifier, each with at least 10 splits in the decision tree. Features marked with ✔ indicate refactoring features, ✘ indicate non-refactoring features — signifies neutral features, and features with the CMV prefix indicate vocabulary features. For the vocabulary features, we provide an example of a commit message containing that feature and set the possible refactoring operation in parentheses. For other features, we include a description of the feature.

Feature Type	Feature Name	Importance	Refactoring	Example / Description
Textual	**CMV:updat (update)**	170	✔	The return type of the method call is updated (Change Return Type)
	CMV: remov (remove)	145	✔	The parameters of the method is/are removed (Remove Parameter)
	CMV:refactor	78	✔	Refactored the contributor guide into many more sub sections and made a start on some of them
	CMV:improv	50	✔	Small name update to improve clarity in func_wrapper (Change Function Name)
	CMV:move	45	✔	Move dtype functions from general to dtype jax (move method)
	CMV:renam (rename)	43	✔	The method is_array is renamed to is_native_array (Rename Method)
	Readability	41	—	The Flesch commit message readability metric [155] (described in Section 5.2.4)
	Words Count	41	—	Number of words in the commit message
	CMV:clean	38	✔	Clean up databricks scripts (Multiple Refactoring Operations)
	CMV:cleanup	32	✔	Fixed typos in code cleanup process (Change Variable Name)
	CMV:add	29	✔	The parameters are added to the method get_relevant_items_by_threshold (Pull Up Parameters)
	CMV:simplifi (simplify)	23	✔	Simplification of predict function through parameter (Remove Parameter)
	CMV:split	22	✔	Split configuration and modeling files (Move Method)
	CMV:unus (unused)	19	✔	Remove unused args parameter from PreTrainedConfig.from_pretrained (Remove Parameter)
	CMV:readme	15	✘	Rename the readme file
	CMV:restructur (restructure)	14	✔	Restructured function (Move Method)
	CMV:option	14	✔	Add options for save_as (Add Parameter)
	CMV:doc	12	✘	Update document of standalone deployment
	CMV:name	12	✔	Rename some function (Rename Method)
	CMV:ad	11	—	Import optional modules ad hoc in the runner
	CMV:paramet (parameter)	10	✔	Change parameter names (Rename Parameter)
	CMV:fix	10	✘	Fix parallel execute for bug
	CMV:argument	10	✔	Removed unused argument (Remove parameter)
Precess	**Refactoring Contribution**	239	✔	The developer refactoring contribution (described in Section 5.2.4)
	Lines Deleted	140	✔	The number of lines deleted in a commit
	Has Executable File	108	—	If a commit contains executable code
	Lines Added	94	✔	The number of lines added in the commit
	Code Entropy	88	✔	The code entropy of each commit (described in Section 5.2.4)
	Lines Changed	49	—	The number of lines changed in a commit (Lines Added - Lines Deleted)
Code	**CountLineCodeDecl**	87	✔	Count of lines that consist of declarative source code [180]
	CountDeclFunction	51	—	Methods declared within a class accessible exclusively via an instance of that class [180]
	SumCyclomatic	42	✔	The total weight of methods belonging to a class [180]
	AvgCountLineCode	36	✔	Mean number of lines containing source code within all nested functions or methods [180]
	CountLineComment	30	✔	Mean number of lines containing comments for all nested functions or methods [180]
	CountLineBlank	24	✔	The number of black lines of code [180]
	RatioCommentToCode	22	—	Proportion of lines containing comments compared to lines containing code [180]
	MaxCyclomatic	19	✔	The highest cyclomatic complexity among all nested functions or methods [180]

keyword, are considered dependent features that assist other features in the decision-making process. They do not directly influence the model's decision but contribute to the overall classification outcome in conjunction with other features. They enhance the model's predictions by adding additional context that complements other features, thereby playing a supportive role. The results of our feature importance approach with at least 10 splits in the decision tree are listed in Table 5.5.

Our approach provides a complementary set of important textual, process, and code features, helping practitioners, researchers, and tool makers understand the significance of different features in identifying refactoring commits.

$RQ_{5.2}$: Summary

Textual features, which capture the characteristics of commit messages, play a crucial role in detecting refactoring commits. Additionally, past refactoring contributions of developers, code changes, and code complexity features significantly contribute to identifying refactoring commits that lack indicators in their commit messages.

5.3.3 $RQ_{5.3}$: Can we leverage commit information to complement existing keyword-based and rule-based approaches?

Motivation

Historical information of software has been used to identify refactorings in prior studies [19, 29–32]. Some approaches use a specific set of keywords to determine whether a commit is related to refactoring or not [29–32]. Moreover, certain tools, like PyRef [19], utilize pattern matching on the AST level to ascertain the presence of refactoring in a commit. In this research question, we first compare our approach with

state-of-the-art tools and methods that identify refactoring commits to evaluate the effectiveness of our approach. Moreover, we aim to determine whether ensembling our approach with PyRef, the state-of-the-art and rule-based approach, aids in improving refactoring commit detection. This analysis helps us provide a more effective solution by complementing the state-of-the-art tools to identify refactoring commits in ML projects mainly written in Python.

Approach

Comparing MLRefScanner with baselines. We compare the output of our approach with the following keyword-based and rule-based approaches by measuring the performance metrics (*i.e.,* precision, recall, F1, and AUC) on the testing set in the mixed-projects setting. For the keyword-based approaches [29–32], state-of-the-art studies have used or identified certain keywords to identify refactoring commits in the commit history. We use the same keywords as the state-of-the-art approaches and search through the entire commit history to identify refactoring commits based on the keywords provided by the state-of-the-art approaches.

For the rule-based approach [19], we run PyRef accordingly to get refactoring commits. As discussed in Section 5.3.3, PyRef achieves the highest precision (100%) compared to its opponents, but it suffers from low recall. This motivates us to consider ensembling our approach with PyRef to potentially improve the low recall of the PyRef.

Ensembling MLRefScanner with PyRef. In our approach, we obtain a best-performing classifier that utilizes code, textual, and process features to identify whether a commit is a refactoring commit or not. Conversely, PyRef employs

pre-defined rules to identify different types of refactoring and, consequently, refactoring commits. To potentially improve the recall in predicting refactoring commits, we propose an ensemble learning [181] method to combine the results from the best-performing classifier (*i.e.,* MLRefScanner) with PyRef and measure if we can complement PyRef and improve its low recall. Using ensembling, we measure each classifier's vote individually and combine their decisions to determine the final label. Therefore, both our classifier and PyRef vote on whether a commit is refactoring or not. We test and apply unanimous (*i.e.,* both PyRef and our approach detect refactoring) and majority (*i.e.,* at least one of PyRef or our approach detects refactoring) voting schemes and measure the enhancements in refactoring commit detection. This provides a more robust and comprehensive decision-making process.

Findings

MLRefScanner outperforms the state-of-the-art rule-based and keyword-based approaches, achieving, on average, 36% higher precision, 52% higher recall, 28% higher AUC, and 30% higher F1 scores compared to previous keyword-based approaches [29–32] and 6% lower precision, 49% higher recall, and 22% higher AUC, compared to the rule-based state-of-the-art approach [19] on the testing set. Table 5.6 shows the results of our approach compared to other refactoring detection approaches on our ML testing set in the mixed-projects setting. MLRefScanner achieves a recall of 82%, surpassing PyRef [19] at 33%, AlOmar *et al.* [29] at 39%, Ratzinger *et al.* [30] at 26%, Zhang *et al.* [31] at 29%, and Kim *et al.* [32] at 25%. This indicates that previous approaches

Table 5.6: Comparison of results obtained from MLRefScanner and state-of-the-art approaches.

Appraoch	**Category**	**Precision**	**Recall**	**AUC**	**F1**
MLRefScanner	ML Based	94%	82%	89%	87%
PyRef	Rule Based	100%	33%	67%	50%
AlOmar *et al.*	Keyword Based	39%	39%	56%	39%
Ratzinger *et al.*	Keyword Based	84%	26%	62%	40%
Zhang *et al.*	Keyword Based	48%	29%	57%	36%
Kim *et al.*	Keyword Based	93%	25%	62%	39%

Table 5.7: Comparison of results obtained from Our classifier, PyRef, and ensemble approaches.

Method	**Precision**	**Recall**	**AUC**	**F1**
PyRef	100%	39%	70%	56%
MLRefScanner	94%	82%	89%	87%
Unanimous Voting	100%	21%	61%	35%
Majority Voting	95%	99%	98%	97%

overlook a significant number of refactoring commits, making MLRefScanner superior to state-of-the-art methods in detecting all possible refactoring commits. ***Our approach can complement the low recall of PyRef with a 60% increase in its recall in the majority vote setting.*** Despite a 6% lower precision, our approach achieves a 49% higher recall compared to PyRef, indicating our approach can capture a broader window of refactoring commits in the development history. Table 5.7 shows our ensembling results by combining our approach and PyRef. Our ensemble approach achieves a precision of 95%, recall of 99%, an AUC of 98%, and an F1 score of 97% in the majority vote setting. Therefore, we can effectively complement and assist PyRef in finding the refactoring commits that might have otherwise been overlooked.

$RQ_{5.3}$: Summary

On average, MLRefScanner provides 21% higher precision, 52% higher recall, 28% higher AUC, and 45% higher F1 score compared to state-of-the-art rule-based and keyword-based approaches. MLRefScanner can complement PyRef's low recall and significantly increase it by 60% in refactoring commit detection, motivating refactoring tool makers to consider more features in the refactoring detection process.

5.4 Discussion on the Generalizability of Our Approach

As discussed in Section 5.3.1, MLRefScanner performs well within ML projects and achieves high precision and recall under different testing settings in the ML technical domain compared to rule-based and keyword-based state-of-the-art approaches. In this section, we want to investigate the potential applicability of MLRefScanner across diverse technical domains by assessing its performance on general (*i.e.,* non-ML) Python projects. For this purpose, we assemble a testing dataset comprising a variety of general Python projects, aiming to evaluate the adaptability of MLRefScanner beyond machine learning applications. We follow similar project selection criteria used for choosing machine learning projects to select the general Python projects (described in Section 5.2.2). We do not restrict the number of commits for general projects, allowing us to evaluate the effectiveness of our approach on both smaller projects with a limited number of commits and larger projects. This process results in a set of 12,665 general projects. Subsequently, we randomly select the same number of projects as in the machine learning category (*i.e.,* 199 projects) to evaluate the performance of our approach in general Python projects. We manually validate the

general Python projects to ensure that projects containing only documentation are not included. Following this, we validate and verify our model to further assess the generalizability in the general Python projects. We assess the performance of our model in the context of general Python projects under the following scenarios:

- **Combined General Projects:** Using a testing dataset comprising 199 general Python projects, we assess the performance of our model trained on ML projects across the entire dataset of general projects. This testing dataset includes all commits from all 199 general projects.

- **Individual General Projects:** Using individual general Python projects as testing datasets, we individually test our model trained using ML projects, on all commits of each general Python project. This approach allows us to assess the effectiveness of our model within the specific context of each project.

MLRefScanner can be extended to projects within other technical domains. In the combined-general-projects setting, our model successfully achieves a precision of 99%, a recall of 93%, and an AUC of 96% on general Python projects. In the individual-general-projects setting, we attain a median precision of 99%, a median recall of 100%, and a median AUC of 100%. Maintaining high performance across technical domains is attributed to our use of features extractable from any Python repository, which are not specific to any particular domain. The performance of our approach can generalize from one technical domain (*i.e.*, ML projects) to other domains (*i.e.,* general Python projects). Practitioners and researchers can apply our findings in various contexts, whether they are working on traditional Python projects or those involving machine learning while maintaining a high level of performance.

5.5 Threats to Validity

In this section, we discuss the threats to the validity of our study.

Threats to Construct Validity. concern the data preparation, feature selection, and model construction methods employed in our study. In dataset preparation, we utilize a set of textual, process, and code features to characterize the commit messages. Nevertheless, incorporating additional features has the potential to further enhance the accuracy of our classification model. In the model construction phase, we utilize random search for hyperparameter tuning in the model construction due to its cost-effectiveness, other hyperparameter tuning approaches could potentially further improve the accuracy of our classifier. To assess the performance of the models across commits with varying numbers of refactoring operations, we conduct a sensitivity analysis by adjusting the labeling threshold of the number of refactoring operations in a commit obtained from PyRef. We measure the performance of the models with different thresholds to evaluate the performance of the models in identifying refactoring commits. As the threshold for refactoring operations increases, the recall of the model drops, meaning that the classifier dismisses the commits with fewer numbers of refactoring operations. This observation further supports the efficacy of the selected threshold (*i.e.,* the existence of at least one refactoring operation in a commit) in identifying refactoring commits. Although we considered a large number of different projects to ensure diversity in the commit message keywords, our automatic labeling approach may sometimes still generate false positives or false negatives, especially when there are ambiguous keywords or synonyms. To understand the validity of our automated labeling, we manually verified the automated labeling results by examining

a statistically representative sample from the entire commit dataset, demonstrating the high precision and recall of our automated labeling approach. This approach was selected because validating all commits was not feasible with our available human resources. However, we acknowledge that this labeling may be potentially imprecise and may not be generalized to specific domains or projects using different commit message conventions.

Threats to Internal Validity. indicate the validity of methods utilized in our study. In the experiment setup, we opt to use term frequency (TF) to vectorize commit messages [182] instead of using other vectorizing approaches, such as Bert [153] and word2vec [154], due to their lack of transparency and irreversibility. While we acknowledge that using such vector transformation algorithms could potentially enhance our commit message analysis, we prioritize the interpretability and reversibility of the chosen approach in order to provide explainability of our approach. In the process of labeling validation and ground truth preparation, we obtain a Cohen's kappa agreement level of 0.64, indicating a moderate level of agreement between the validators. However, the two validators have discussed the results of the manual validations meticulously to reach a conclusion on each commit and achieve a consensus on the final label, ensuring the avoidance of bias in the labeling process. In comparing our approach with state-of-the-art keyword-based approachess [29–32], we utilized the set of keywords provided by these approaches to detect refactoring-related commits. However, we acknowledge that the recommended keywords from these state-of-the-art approaches may be specifically relevant to their studied datasets.

Threats to External Validity. indicate the generalizability of our approach. The

textual features that our classification model is trained on are tailored to English languages. The reported performance scores reflect the performance of our classification model on repositories that predominantly use English to document their commits. As a result, when applied to non-English repositories, the performance of our approach may decrease. We validate our approach on 199 general English Python projects to ensure its generalizability across various Python projects. Since our labeling and machine learning approaches rely on the quality of the commit messages written by developers, the performance of our ML-based approach and our evaluation results may be challenged if developers do not use informative commit messages, potentially leading to incorrect labels and prediction results in some cases.

5.6 Summary

In this chapter, we introduce our approach, MLRefScanner, which is capable of identifying refactoring commits in the history of ML projects. We conduct extensive experiments on 199 ML Python projects and verify the performance of MLRefScanner in various testing scenarios. MLRefScanner achieves a precision of 94%, a recall of 82%, and an AUC of 89%, and a minimum set of features is provided for distinguishing refactoring commits. Our approach outperforms state-of-the-art refactoring detection methods, with an average of 21% higher precision, 52% higher recall, 28% higher AUC, and 45% higher F1 scores in the mixed-projects scenario. Moreover, ensembling MLRefScanner with PyRef, the most recent and accurate refactoring detection tool, showcases a 60% improvement in its overall recall. MLRefScanner successfully identifies refactoring activities that were previously overlooked by state-of-the-art methods and can detect commits involving ML-specific refactoring operations. MLRefScanner

enhances the refactoring detection in ML technical domain by addressing the issue of low recall present in existing state-of-the-art approaches. Moreover, we contribute to the refactoring research community with an extensive dataset that includes refactoring activities from 199 ML repositories for further refactoring analysis. In the future, we plan to extend and fine-tune our approach to enable the detection of specific types of refactorings, such as pull-up method and move-class, by mining software development histories.

Chapter 6

Conclusion and Future Work

This thesis addresses the current gap in large-scale studies on refactoring practices, focusing on refactoring rhythms, tactics, and release-wise refactoring patterns. Additionally, by measuring an extensive set of code quality metrics, it explores the relationship between the identified practices with code quality. The thesis also proposes an ML-based refactoring detection approach for detecting refactoring-related commits in Python programming language within the ML domain. This approach is used to investigate refactoring operations in both Python and ML technical domains, to advance future studies on refactoring in non-Java programming languages.

To achieve this, we conduct three empirical studies: (1) a study on refactoring rhythms and tactics, (2) a study on release-wise refactoring patterns, and (3) a study on the detection of refactoring commits in Python projects.

6.1 Contributions

The main contributions of this thesis are as follows.

- **Propose a catalog of refactoring rhythms and tactics used by developers and highlight their relationship with code quality.** By applying statistical tests and time series clustering methods, we identify and cluster weekly and long-term refactoring activities into distinct patterns, referred to as refactoring rhythms and tactics. Furthermore, by measuring code smells before and after each refactoring rhythm or tactic and assessing the significance and magnitude of changes, we establish the relationship between each rhythm or tactic and code quality.

  Summary of Chapter 3

  Our results classify refactoring rhythms into two categories: *all-day* and *work-day* rhythms. We further classify refactoring tactics into four types: *intermittent root canal, intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. Regarding the relationship between rhythms and code quality, we did not observe significant differences in code quality following the application of either *work-day* or *all-day* rhythms. However, the two root canal-based tactics rank highest in reducing the overall frequency of code smells, particularly for 28 out of 35 types of code smells. Root canal-based tactics generally lead to a decrease in the frequency of code smells, whereas floss-based tactics tend to result in an increase. Consequently, root canal-based tactics outperform floss-based tactics in reducing the total number of code smells.

- **Identify common release-wise refactoring patterns within release cycles and their relationship with code quality.** By analyzing refactoring densities within release cycles in CI/CD and applying time series clustering techniques, we identify four common release-wise refactoring patterns. Using

different code quality metrics, we assess the relationship between each identified pattern and code quality. Additionally, we explore the distribution of the release-wise refactoring patterns across different stages of the project lifecycle and evaluate how transitions between patterns impact code quality.

Summary of Chapter 4

We identify four dominant release-wise refactoring patterns: *early inactive*, *early active*, *steady active*, and *steady inactive*. Among these, the *early inactive* pattern demonstrates the best performance in reducing coupling, increasing cohesion, and minimizing code smells. Although *steady active* pattern performs well on code quality metrics and total code smells, it exhibits the worst performance in reducing architectural code smells, suggesting that excessive refactoring may expose more architectural smells. The usage of the *steady active* pattern increases as projects progress; however, *late active* may be a better alternative, which is the most reliable release-wise refactoring pattern, remains consistent throughout the project lifecycle.

- **Propose a generalizable machine-learning-based prototype tool for detecting refactoring commits in Python projects within the ML technical domain and demonstrate how ensembling our approach with state-of-the-art methods can enhance their performance.** By utilizing a wide range of process, textual, and code features of commits, we develop ML-RefScanner, a tool capable of identifying refactoring-related commits in Python projects. Additionally, we investigate the key features and ML-specific refactoring operations of these commits, providing insights to help practitioners understand and expand refactoring-related studies in programming languages such as

Python.

Summary of Chapter 5

MLRefScanner achieves an overall precision of 94% and a recall of 82%. Compared to state-of-the-art rule-based and keyword-based approaches, MLRefScanner offers 21% higher precision, 52% higher recall, 28% higher AUC, and a 45% higher F1 score. Additionally, MLRefScanner can significantly complement PyRef's low recall, increasing it by 60% in refactoring commit detection. This highlights the importance of considering a broader range of features in the refactoring detection process, providing valuable insights for refactoring tool developers. Furthermore, we investigate ML-specific refactoring operations that are overlooked by existing refactoring detection tools.

6.2 Future Work

The results of this thesis demonstrate the significant role of refactoring in development practices and their relationship with code quality. It also explores new dimensions of refactoring detection in Python programming language. Future work can build on the findings of this thesis by exploring various perspectives in code refactoring research.

Cross-language refactoring detection. In Chapter 5, we introduce MLRefScanner, which is capable of detecting refactoring-related commits in project histories by utilizing a set of metrics overlooked by state-of-the-art approaches. Future work can build on this by using these metrics or developing a set of language-independent metrics to create a more flexible cross-language refactoring detection tool.

Expanding refactoring type coverage in tools. Existing refactoring detection

tools that can identify refactoring operations within code history are primarily written for Java. Future work should focus on enhancing these tools or developing new ones for popular languages like Python, aiming to achieve the same level of coverage as state-of-the-art tools for Java.

Integration of code quality assessment into the CI/CD pipeline. Our results highlight significant differences in release quality associated with various release-wise refactoring patterns. Future work could use these findings as a baseline to develop and integrate an automated quality assessment component into the release pipeline.

Enhance automatic refactoring for IDEs. Future research could further enhance IDE refactoring recommendations by providing more detailed information on optimal refactoring timings and specific types of refactorings that can improve code quality.

User-oriented refactoring recommendation tools. Our results highlight the importance of adopting specific refactoring patterns and the influence of previous refactoring activities of developers on their future refactorings. Future studies can explore personalized refactoring recommendations based on a developer's refactoring history.

Leveraging large language models. Future research could explore the potential of large language models (LLMs) in refactoring detection, automating code refactoring, and providing code refactoring recommendations.

Bibliography

[1] G. Moser, R. Vallon, M. Bernhart, and T. Grechenig, "Teaching software quality assurance with gamification and continuous feedback techniques," in 2021 IEEE Global Engineering Education Conference (EDUCON). IEEE, 2021, pp. 505–509.

[2] J. Börstler, K. E. Bennin, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner, B. MacKellar, R. Duran, H. Störrle, D. Toll et al., "Developers talking about code quality," Empirical Software Engineering, vol. 28, no. 6, p. 128, 2023.

[3] I. O. for Standardization, Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models. ISO, 2011.

[4] P. Tomas, M. J. Escalona, and M. Mejias, "Open source tools for measuring the internal quality of java software products. a survey," Computer Standards & Interfaces, vol. 36, no. 1, pp. 244–255, 2013.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing code addison-wesley professional," Berkeley, CA, USA, 1999.

[6] J. Kerievsky, "Refactoring to patterns (addison-wesley signature series)," 2005.

[7] T. Sharma, "Quantifying quality of software design to measure the impact of refactoring," in 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops. IEEE, 2012, pp. 266–271.

[8] R. C. Martin, Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009.

[9] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in Proceedings of the 2017 11th Joint Meeting on foundations of Software Engineering, 2017, pp. 465–475.

[10] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," IEEE Transactions on Software Engineering, 2020.

[11] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 2015, pp. 253–258.

[12] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, "Revisiting the relationship between code smells and refactoring," in 2016 IEEE 24th International Conference on Program Comprehension (ICPC). IEEE, 2016, pp. 1–4.

[13] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, "A quantitative study on characteristics and effect of batch refactoring on code smells," in 2019 ACM/IEEE

International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2019, pp. 1–11.

[14] 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.

[15] 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.

[16] A. Martini and J. Bosch, "An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt," in 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C). IEEE, 2016, pp. 31–40.

[17] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11.

[18] B. Du Bois, S. Demeyer, and J. Verelst, "Does the" refactor to understand" reverse engineering pattern improve program comprehension?" in Ninth european conference on software maintenance and reengineering. IEEE, 2005, pp. 334–343.

[19] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza, "Pyref: refactoring detection in python projects," in 2021 IEEE 21st international working conference on source code analysis and manipulation (SCAM). IEEE, 2021, pp. 136–141.

[20] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021, pp. 348–357.

[21] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 483–494.

[22] S. Mihajlović, A. Kupusinac, D. Ivetić, and I. Berković, "The use of python in the field of artifical intelligence," in International Conference on Information Technology and Development of Education–ITRO, 2020.

[23] K. Srinath, "Python–the fastest growing programming language," International Research Journal of Engineering and Technology, vol. 4, no. 12, pp. 354–357, 2017.

[24] Aug 2024. [Online]. Available: https://www.tiobe.com/tiobe-index/

[25] S. Raschka, J. Patterson, and C. Nolet, "Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence," Information, vol. 11, no. 4, p. 193, 2020.

[26] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D.-R. Roberts, "Improving the design of existing code," Addisonâ˜Wesley, vol. 6, 1999.

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

[28] L. Tan and C. Bockisch, "A survey of refactoring detection tools." in Software Engineering (Workshops), 2019, pp. 100–105.

[29] E. AlOmar, M. W. Mkaouer, and A. Ouni, "Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages," in 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR). IEEE, 2019, pp. 51–58.

[30] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining software evolution to predict refactoring," in First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007). IEEE, 2007, pp. 354–363.

[31] Z. Di, B. Li, Z. Li, and P. Liang, "A preliminary investigation of self-admitted refactorings in open source software (s)," in International Conferences on Software Engineering and Knowledge Engineering, vol. 2018. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018, pp. 165–168.

[32] M. Kim, T. Zimmermann, w. a. t. e. g. i. u. r. p. b. a. c. r. t. u. b. d. Nagappan, NachiIn this thesis, . l.-t. r. t. their relationship with code quality. These practices include (1) weekly refactoring practices, and i. P. (3) release-wise refactoring patterns. Furthermore, we extend our study to refactoring practices in machine learning (ML) projects using Python—a prominent domain for this language—to guide future research toward less extensively studied languages, "An empirical study of refactoringchallenges and benefits at microsoft," IEEE Transactions on Software Engineering, vol. 40, no. 7, pp. 633–649, 2014.

[33] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," IEEE Transactions on Software Engineering, vol. 39, no. 8, pp. 1144–1156, 2012.

[34] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in Proceedings of 28th IEEE international conference on software maintenance (ICSM). IEEE, 2012, pp. 306–315.

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

[36] L. Aversano, U. Carpenito, and M. Iammarino, "An empirical study on the evolution of design smells," Information, vol. 11, no. 7, p. 348, 2020.

[37] T. Sharma, "Multi-faceted code smell detection at scale using designitejava 2.0," in 21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024. ACM, 2024, pp. 284–288. [Online]. Available: https://doi.org/10.1145/3643991.3644881

[38] M. Ó Cinnéide, A. Yamashita, and S. Counsell, "Measuring refactoring benefits: a survey of the evidence," in Proceedings of the 1st International Workshop on Software Refactoring, 2016, pp. 9–12.

[39] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in International conference on software reuse. Springer, 2006, pp. 287–297.

[40] 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, 2016, pp. 858–870.

[41] J. Ivers, I. Ozkaya, R. L. Nord, and C. Seifried, "Next generation automated software evolution refactoring at scale," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1521–1524.

[42] E. Soares, M. Ribeiro, G. Amaral, R. Gheyi, L. Fernandes, A. Garcia, B. Fonseca, and A. Santos, "Refactoring test smells: A perspective from open-source developers," in Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing, 2020, pp. 50–59.

[43] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," IEEE software, vol. 25, no. 5, pp. 38–44, 2008.

[44] L. Sousa, W. Oizumi, A. Garcia, A. Oliveira, D. Cedrim, and C. Lucena, "When are smells indicators of architectural refactoring opportunities: A study

of 50 software projects," in Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 354–365.

[45] T. T. Klooster, "Effectiveness and scalability of fuzzing techniques in ci/cd pipelines," Ph.D. dissertation, 2021.

[46] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study," in 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2021, pp. 471–482.

[47] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 426–437.

[48] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of mozilla firefox," in 2012 9th IEEE working conference on mining software repositories (MSR). IEEE, 2012, pp. 179–188.

[49] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou, "Understanding the impact of rapid releases on software quality: The case of firefox," Empirical Software Engineering, vol. 20, pp. 336–373, 2015.

[50] T. Jansen, Z. Abou Khalil, E. Constantinou, and T. Mens, "Does the duration of rapid release cycles affect the bug handling activity?" in 2021 IEEE/ACM

4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal). IEEE, 2021, pp. 17–24.

[51] B. Adams and S. McIntosh, "Modern release engineering in a nutshell–why researchers should care," in 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol. 5. IEEE, 2016, pp. 78–90.

[52] R. Baskerville and J. Pries-Heje, "Short cycle time systems development," Information Systems Journal, vol. 14, no. 3, pp. 237–264, 2004.

[53] Aug 2024. [Online]. Available: https://www.python.org/doc/essays/blurb/

[54] S. Khoirom, M. Sonia, B. Laikhuram, J. Laishram, and T. D. Singh, "Comparative analysis of python and java for beginners," Int. Res. J. Eng. Technol, vol. 7, no. 8, pp. 4384–4407, 2020.

[55] S. A. Abdulkareem and A. J. Abboud, "Evaluating python, c++, javascript and java programming languages based on software complexity calculator (halstead metrics)," in IOP Conference Series: Materials Science and Engineering, vol. 1076, no. 1. IOP Publishing, 2021, p. 012046.

[56] J. Zhang, Y. Chen, Q. Gong, X. Wang, A. Y. Ding, Y. Xiao, and P. Hui, "Understanding the working time of developers in it companies in china and the united states," IEEE Software, vol. 38, no. 2, pp. 96–106, 2020.

[57] M. Claes, M. V. Mäntylä, M. Kuutila, and B. Adams, "Do programmers work at night or during the weekend?" in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 705–715.

[58] X. Wang, S. Xu, L. Peng, Z. Wang, C. Wang, C. Zhang, and X. Wang, "Exploring scientists' working timetable: Do scientists often work overtime?" Journal of Informetrics, vol. 6, no. 4, pp. 655–660, 2012.

[59] C. Binnewies, S. Sonnentag, and E. J. Mojza, "Recovery during the weekend and fluctuations in weekly job performance: A week-level study examining intra-individual relationships," Journal of Occupational and Organizational Psychology, vol. 83, no. 2, pp. 419–441, 2010.

[60] H. Liu, Y. Gao, and Z. Niu, "An initial study on refactoring tactics," in 2012 IEEE 36th Annual Computer Software and Applications Conference. IEEE, 2012, pp. 213–218.

[61] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," IEEE Transactions on Software Engineering, vol. 47, no. 12, pp. 2786–2802, 2020.

[62] I. H. Moghadam, M. Ó. Cinnéide, F. Zarepour, and M. A. Jahanmir, "Refdetect: A multi-language refactoring detection tool based on string alignment," IEEE Access, vol. 9, pp. 86 698–86 727, 2021.

[63] M. K. Shiblu, "Jsdiffer: Refactoring detection in javascript," Ph.D. dissertation, Concordia University Montréal, Québec, Canada, 2022.

[64] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, 2010, pp. 371–372.

[65] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," Journal of Systems and Software, vol. 86, no. 4, pp. 1006–1022, 2013.

[66] P. S. Sagar, E. A. AlOmar, M. W. Mkaouer, A. Ouni, and C. D. Newman, "Comparing commit messages and source code metrics for the prediction refactoring activities," Algorithms, vol. 14, no. 10, p. 289, 2021.

[67] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, "Refbot: intelligent software refactoring bot," in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, pp. 823–834.

[68] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering repetitive code changes in python ml systems," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 736–748.

[69] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007). IEEE, 2007, pp. 10–10.

[70] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Toward the automatic classification of self-affirmed refactoring," Journal of Systems and Software, vol. 171, p. 110821, 2021.

[71] A. Almogahed, M. Omar, and N. H. Zakaria, "Impact of software refactoring on software quality in the industrial environment: A review of empirical studies," 2018.

[72] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," Journal of Systems and Software, vol. 167, p. 110610, 2020.

[73] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, "On experimenting refactoring tools to remove code smells," in Scientific Workshop Proceedings of the XP2015, 2015, pp. 1–8.

[74] N. Baumgartner, P. Iyenghar, T. Schoemaker, and E. Pulvermüller, "Ai-driven refactoring: A pipeline for identifying and correcting data clumps in git repositories," Electronics, vol. 13, no. 9, p. 1644, 2024.

[75] I. Saidani, A. Ouni, M. W. Mkaouer, and F. Palomba, "On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent," Information and Software Technology, vol. 138, p. 106618, 2021.

[76] Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?" in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1435–1446.

[77] L. Chen and M. A. Babar, "Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development," in 2014 IEEE/IFIP Conference on Software Architecture. IEEE, 2014, pp. 195–204.

[78] V. Khorikov, "Short-term vs long-term perspective in software development." [Online]. Available: https://enterprisecraftsmanship.com/posts/short-term-vs-long-term-perspective/

[79] M. Claes and M. V. Mäntylä, "20-mad: 20 years of issues and commits of mozilla and apache development," in Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 503–507.

[80] berkeley bootcamp, "11 Most In-Demand Programming Languages in 2021," Dec. 2020. [Online]. Available: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

[81] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen, "An empirical study of design degradation: How software projects get worse over time," in 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 2015, pp. 1–10.

[82] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, 2nd ed. New York: Routledge, Jul. 1988.

[83] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities, 2016, pp. 1–4.

[84] T. Sharma and D. Spinellis, "A survey on software smells," Journal of Systems and Software, vol. 138, pp. 158–173, 2018.

[85] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in 2017

IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017, pp. 282–285.

[86] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in 2015 12th Working IEEE/IFIP Conference on Software Architecture. IEEE, 2015, pp. 51–60.

[87] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in 2008 12th European conference on software maintenance and reengineering. IEEE, 2008, pp. 329–331.

[88] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE, 2019, pp. 88–97.

[89] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in 2009 13th European Conference on Software Maintenance and Reengineering. IEEE, 2009, pp. 255–258.

[90] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 740–740.

[91] AlDanial, "AlDanial/cloc," Jul. 2021, original-date: 2015-09-07T03:30:43Z. [Online]. Available: https://github.com/AlDanial/cloc

[92] GitHub, "GitHub API," https://docs.github.com/en/rest, accessed: April 3, 2023.

[93] F. E. Harrell et al., Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis. Springer, 2015, vol. 3.

[94] E. Noei, F. Zhang, and Y. Zou, "Too many user-reviews, what should app developers look at first?" IEEE Transactions on Software Engineering, 2019.

[95] J. Miles, "R-squared, adjusted r-squared," Encyclopedia of statistics in behavioral science, 2005.

[96] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 182–191.

[97] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," Journal of the royal statistical society. series c (applied statistics), vol. 28, no. 1, pp. 100–108, 1979.

[98] R. L. Thorndike, "Who belongs in the family?" Psychometrika, vol. 18, no. 4, pp. 267–276, 1953.

[99] M. Syakur, B. Khotimah, E. Rochman, and B. D. Satoto, "Integration k-means clustering method and elbow method for identification of the best customer profile cluster," in IOP Conference Series: Materials Science and Engineering, vol. 336, no. 1. IOP Publishing, 2018, p. 012017.

[100] P. E. McKnight and J. Najab, "Mann-whitney u test," The Corsini encyclopedia of psychology, pp. 1–1, 2010.

[101] W. Kruskal, "Kruskall–wallis one way analysis of variance," J Am Stat Assoc, vol. 47, no. 260, pp. 583–621, 1952.

[102] T. Dahiru, "P-value, a true test of statistical significance? a cautionary note," Annals of Ibadan postgraduate medicine, vol. 6, no. 1, pp. 21–26, 2008.

[103] S. K. Haldar, "Statistical and geostatistical applications in geology," Mineral exploration, pp. 167–194, 2018.

[104] L. Rising and N. S. Janoff, "The scrum software development process for small teams," IEEE software, vol. 17, no. 4, pp. 26–32, 2000.

[105] J. C. Gower, "Properties of euclidean and non-euclidean distance matrices," Linear algebra and its applications, vol. 67, pp. 81–97, 1985.

[106] M. Kljun and M. Ters˘ek, "A review and comparison of time series similarity measures," in 29th International Electrotechnical and Computer Science Conference (ERK 2020). Portoroz˘, 2020, pp. 21–22.

[107] S. K. Gaikwad, B. W. Gawali, and P. Yannawar, "A review on speech recognition technique," International Journal of Computer Applications, vol. 10, no. 3, pp. 16–24, 2010.

[108] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," IEEE Transactions on Software Engineering, vol. 43, no. 1, pp. 1–18, 2016.

[109] ——, "The impact of automated parameter optimization on defect prediction models," IEEE Transactions on Software Engineering, vol. 45, no. 7, pp. 683–711, 2018.

[110] 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, no. 1, pp. 3–17, 2014.

[111] C. J. Ferguson, "An effect size primer: a guide for clinicians and researchers." 2016.

[112] I. Spieler, S. Scheibe, C. Stamov-Roßnagel, and A. Kappas, "Help or hindrance? day-level relationships between flextime use, work–nonwork boundaries, and affective well-being." Journal of Applied Psychology, vol. 102, no. 1, p. 67, 2017.

[113] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in Annual meeting of the Florida association of institutional research, 2006.

[114] "Scikit-learn k-means silhouette analysis," https://scikit-learn.org/stable/auto_examples/
cluster/plot_kmeans_silhouette_analysis.html, accessed: August 23, 2023.

[115] E. S. Dalmaijer, C. L. Nord, and D. E. Astle, "Statistical power for cluster analysis," BMC bioinformatics, vol. 23, no. 1, pp. 1–28, 2022.

[116] U. Rani and S. Sahu, "Comparison of clustering techniques for measuring similarity in articles," in 2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT). IEEE, 2017, pp. 1–7.

[117] T. Yatsunenko, F. E. Rey, M. J. Manary, I. Trehan, M. G. Dominguez-Bello, M. Contreras, M. Magris, G. Hidalgo, R. N. Baldassano, A. P. Anokhin et al., "Human gut microbiome viewed across age and geography," nature, vol. 486, no. 7402, pp. 222–227, 2012.

[118] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," Journal of computational and applied mathematics, vol. 20, pp. 53–65, 1987.

[119] T. L. Wahl, "Discussion of "despiking acoustic doppler velocimeter data" by derek g. goring and vladimir i. nikora," Journal of Hydraulic Engineering, vol. 129, no. 6, pp. 484–487, 2003.

[120] M. Parsheh, F. Sotiropoulos, and F. Porté-Agel, "Estimation of power spectra of acoustic-doppler velocimetry data contaminated with intermittent spikes," Journal of Hydraulic Engineering, vol. 136, no. 6, pp. 368–378, 2010.

[121] R. Q. Quiroga, Z. Nadasdy, and Y. Ben-Shaul, "Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering," Neural computation, vol. 16, no. 8, pp. 1661–1687, 2004.

[122] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi, "Refactoring effect on internal quality attributes: What haven't they told you yet?" Information and Software Technology, vol. 126, p. 106347, 2020.

[123] pypl, "PYPL PopularitY of Programming Language index," [Accessed 30-05-2024]. [Online]. Available: https://pypl.github.io/PYPL.html/

[124] S. Noei, H. Li, S. Georgiou, and Y. Zou, "An empirical study of refactoring rhythms and tactics in the software development process," IEEE Transactions on Software Engineering, vol. 49, no. 12, pp. 5103–5119, 2023.

[125] T. Preston-Werner, "Semantic Versioning 2.0.0," [Accessed 30-05-2024]. [Online]. Available: https://semver.org/

[126] S. Phillips, J. Sillito, and R. Walker, "Branching and merging: an investigation into current version control practices," in Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering, 2011, pp. 9–15.

[127] S. Phillips, G. Ruhe, and J. Sillito, "Information needs for integration decisions in the release process of large-scale parallel development," in Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, 2012, pp. 1371–1380.

[128] R. Krasniqi and J. Cleland-Huang, "Enhancing source code refactoring detection with explanations from commit messages," in 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020, pp. 512–516.

[129] T. Sharma, "Designitejava (enterprise)," Sep. 2019, http://www.designite-tools.com/designitejava. [Online]. Available: https://doi.org/10.5281/zenodo.3401802

[130] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in Proceedings of the 1st International Workshop on Bringing

Architectural Design Thinking into Developers' Daily Activities, ser. BRIDGE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–4. [Online]. Available: https://doi.org/10.1145/2896935.2896938

[131] H. A. Reijers and I. T. Vanderfeesten, "Cohesion and coupling metrics for workflow process design," in International conference on business process management. Springer, 2004, pp. 290–305.

[132] SciTools, "Understand: The software developer's multi-tool," https://scitools.com/, [Accessed 12-Jul-2024].

[133] T. J. McCabe, "A complexity measure," IEEE Transactions on software Engineering, no. 4, pp. 308–320, 1976.

[134] T. Klooster, F. Turkmen, G. Broenink, R. Ten Hove, and M. Böhme, "Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in ci/cd pipelines," in 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT). IEEE, 2023, pp. 25–32.

[135] M. Lepot, J.-B. Aubin, and F. H. Clemens, "Interpolation in time series: An introductive overview of existing methods, their performance criteria and uncertainty assessment," Water, vol. 9, no. 10, p. 796, 2017.

[136] M. Friedman, "The interpolation of time series by related series," Journal of the American Statistical Association, vol. 57, no. 300, pp. 729–757, 1962.

[137] H. Akima, "A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points," ACM Transactions on Mathematical Software (TOMS), vol. 4, no. 2, pp. 148–159, 1978.

[138] M. Cuturi and M. Blondel, "Soft-dtw: a differentiable loss function for time-series," in International conference on machine learning. PMLR, 2017, pp. 894–903.

[139] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." Journal of machine learning research, vol. 9, no. 11, 2008.

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

[141] J. R. Norris, Markov chains. Cambridge university press, 1998, no. 2.

[142] M. Dilhara, A. Ketkar, and D. Dig, "Understanding software-2.0: A study of machine learning library usage and evolution," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 4, pp. 1–42, 2021.

[143] M. N. Gevorkyan, A. V. Demidova, T. S. Demidova, and A. A. Sobolev, "Review and comparative analysis of machine learning libraries for machine learning," Discrete and Continuous Models and Applied Computational Science, vol. 27, no. 4, pp. 305–315, 2019.

[144] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" IEEE Transactions on Software Engineering, vol. 47, no. 9, pp. 1857–1871, 2019.

[145] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in 2019 IEEE/ACM 41st International Conference on Software

Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2019, pp. 291–300.

[146] E. de Souza Nascimento, I. Ahmed, E. Oliveira, M. P. Palheta, I. Steinmacher, and T. Conte, "Understanding development process of machine learning systems: Challenges and solutions," in 2019 acm/ieee international symposium on empirical software engineering and measurement (esem). IEEE, 2019, pp. 1–6.

[147] M. T. Ribeiro, S. Singh, and C. Guestrin, "" why should i trust you?" explaining the predictions of any classifier," in Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, 2016, pp. 1135–1144.

[148] M. Openja, F. Majidi, F. Khomh, B. Chembakottu, and H. Li, "Studying the practices of deploying machine learning projects on docker," in Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering, 2022, pp. 190–200.

[149] M. L. McHugh, "Interrater reliability: the kappa statistic," Biochemia medica, vol. 22, no. 3, pp. 276–282, 2012.

[150] R. Thomas. [Online]. Available: https://abiword.github.io/enchant/

[151] V. Balakrishnan and E. Lloyd-Yemoh, "Stemming and lemmatization: A comparison of retrieval performances," 2014.

[152] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," Journal of documentation, vol. 28, no. 1, pp. 11–21, 1972.

[153] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.

[154] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies, 2013, pp. 746–751.

[155] R. Flesch, "A new readability yardstick." Journal of applied psychology, vol. 32, no. 3, p. 221, 1948.

[156] C. E. Shannon, "A mathematical theory of communication," The Bell system technical journal, vol. 27, no. 3, pp. 379–423, 1948.

[157] A. Hamou-Lhadj, "Measuring the complexity of traces using shannon entropy," in Fifth International Conference on Information Technology: New Generations (ITNG 2008). IEEE, 2008, pp. 489–494.

[158] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Trustworthiness models to categorize and prioritize code for security improvement," Journal of Systems and Software, vol. 198, p. 111621, 2023.

[159] A. Zhou, K. Z. Sultana, and B. K. Samanthula, "Investigating the changes in software metrics after vulnerability is fixed," in 2021 IEEE International Conference on Big Data (Big Data). IEEE, 2021, pp. 5658–5663.

[160] N. Vatanapakorn, C. Soomlek, and P. Seresangtakul, "Python code smell detection using machine learning," in 2022 26th International Computer Science and Engineering Conference (ICSEC). IEEE, 2022, pp. 128–133.

[161] F. E. Harrell et al., Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer, 2001, vol. 608.

[162] T. H. Nguyen, B. Adams, and A. E. Hassan, "Studying the impact of dependency network measures on software quality," in 2010 IEEE International Conference on Software Maintenance. IEEE, 2010, pp. 1–10.

[163] J. T. Pintas, L. A. Fernandes, and A. C. B. Garcia, "Feature selection methods for text classification: a systematic literature review," Artificial Intelligence Review, vol. 54, no. 8, pp. 6149–6200, 2021.

[164] J. Jiarpakdee, C. Tantithamthavorn, A. Ihara, and K. Matsumoto, "A study of redundant metrics in defect prediction datasets," in 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2016, pp. 51–52.

[165] M. Hazewinkel, "Minimax principle, encyclopaedia of mathematics," 2001.

[166] S. Kotsiantis, D. Kanellopoulos, P. Pintelas et al., "Handling imbalanced datasets: A review," GESTS international transactions on computer science and engineering, vol. 30, no. 1, pp. 25–36, 2006.

[167] C. Drummond, R. C. Holte et al., "C4. 5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling," in Workshop on learning from imbalanced datasets II, vol. 11, 2003, pp. 1–8.

[168] S. Mishra, "Handling imbalanced data: Smote vs. random undersampling," Int. Res. J. Eng. Technol, vol. 4, no. 8, pp. 317–320, 2017.

[169] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," ACM SIGKDD explorations newsletter, vol. 6, no. 1, pp. 20–29, 2004.

[170] I. Mani and I. Zhang, "knn approach to unbalanced data distributions: a case study involving information extraction," in Proceedings of workshop on learning from imbalanced datasets, vol. 126, no. 1. ICML, 2003, pp. 1–7.

[171] I. Dokmanic, R. Parhizkar, J. Ranieri, and M. Vetterli, "Euclidean distance matrices: essential theory, algorithms, and applications," IEEE Signal Processing Magazine, vol. 32, no. 6, pp. 12–30, 2015.

[172] L. Morán-Fernández, V. Bólon-Canedo, and A. Alonso-Betanzos, "How important is data quality? best classifiers vs best features," Neurocomputing, vol. 470, pp. 365–375, 2022.

[173] H. Jafarzadeh, M. Mahdianpari, E. Gill, F. Mohammadimanesh, and S. Homayouni, "Bagging and boosting ensemble classifiers for classification of multispectral, hyperspectral and polsar data: a comparative evaluation," Remote Sensing, vol. 13, no. 21, p. 4405, 2021.

[174] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." Journal of machine learning research, vol. 13, no. 2, 2012.

[175] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Feature requests-based recommendation of software refactorings," Empirical Software Engineering, vol. 25, pp. 4315–4347, 2020.

[176] G. A. Armijo and V. V. de Camargo, "Refactoring recommendations with machine learning," in Anais Estendidos do XXI Simpósio Brasileiro de Qualidade de Software. SBC, 2022, pp. 15–22.

[177] D. J. Hand, "Assessing the performance of classification methods," International Statistical Review, vol. 80, no. 3, pp. 400–414, 2012.

[178] A. Ghimire, A. K. Jha, S. Thapa, S. Mishra, and A. M. Jha, "Machine learning approach based on hybrid features for detection of phishing urls," in 2021 11th International Conference on Cloud Computing, Data Science & Engineering (Confluence). IEEE, 2021, pp. 954–959.

[179] A. B. Fonseca, D. C. Martins-Jr, Z. Wicik, M. Postula, and S. N. Simões, "Addressing classification on highly imbalanced clinical datasets," in International Conference on Computational Advances in Bio and Medical Sciences. Springer, 2021, pp. 103–114.

[180] Jun 2022. [Online]. Available: https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have-

[181] L. Rokach, "Ensemble-based classifiers," Artificial intelligence review, vol. 33, pp. 1–39, 2010.

[182] H. P. Luhn, "A statistical approach to mechanized encoding and searching of literary information," IBM Journal of research and development, vol. 1, no. 4, pp. 309–317, 1957.

[183] "xkcd: Thesis defense," https://xkcd.com/1403/, (Accessed on 10/20/2017).

[184] "Big bang - warm kitty, soft kitty (sheldon's lullaby sick song) instrumental version lyrics — metrolyrics," http://www.metrolyrics.com/warm-kitty-soft-kitty-sheldons-lullaby-sick-song-instrumental-version-lyrics-big-bang.html?ModPagespeed=noscript, (Accessed on 10/20/2017).

[185] M. Shaw, "Writing good software engineering research papers," in Software Engineering, 2003. Proceedings. 25th International Conference on. IEEE, 2003, pp. 726–736.

[186] B. Paltridge, "Thesis and dissertation writing: an examination of published advice and actual practice," English for Specific Purposes, vol. 21, no. 2, pp. 125–143, 2002.

[187] U. Eco, How to write a thesis. MIT Press, 2015.

[188] R. Rosenthal, H. Cooper, L. Hedges et al., "Parametric measures of effect size," The handbook of research synthesis, vol. 621, no. 2, pp. 231–244, 1994.

[189] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 462–489, 2014.

[190] G. Canfora, L. Cerulo, and M. Di Penta, "On the use of line co-change for identifying crosscutting concern code," in 2006 22nd IEEE International Conference on Software Maintenance. IEEE, 2006, pp. 213–222.

[191] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in Architectures for Adaptive Software Systems: 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24-26, 2009 Proceedings 5. Springer, 2009, pp. 146–162.

[192] G. Samarthyam, G. Suryanarayana, and T. Sharma, "Refactoring for software architecture smells," in Proceedings of the 1st International Workshop on Software Refactoring, 2016, pp. 1–4.

[193] R. Shatnawi and W. Li, "An empirical assessment of refactoring impact on software quality using a hierarchical quality model," International Journal of Software Engineering and Its Applications, vol. 5, no. 4, pp. 127–149, 2011.

[194] P. Arabie, N. D. Baier, C. F. Critchley, and M. Keynes, "Studies in classification, data analysis, and knowledge organization," 2006.

[195] A. Sardá-Espinosa, "Comparing time-series clustering algorithms in r using the dtwclust package," R package vignette, vol. 12, p. 41, 2017.

[196] 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, 2011.

[197] A. Arif and Z. A. Rana, "Refactoring of code to remove technical debt and reduce maintenance effort," in 2020 14th International Conference on Open Source Systems and Technologies (ICOSST). IEEE, 2020, pp. 1–7.

[198] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," IEEE Transactions on Software Engineering, vol. 44, no. 1, pp. 44–69, 2017.

[199] M. Kaya, S. Conley, Z. S. Othman, and A. Varol, "Effective software refactoring process," in 2018 6th International Symposium on Digital Forensic and Security (ISDFS). IEEE, 2018, pp. 1–6.

[200] tiobe, "index | TIOBE - The Software Quality Company." [Online]. Available: https://www.tiobe.com/tiobe-index/

[201] A. M. Mowad, H. Fawareh, and M. A. Hassan, "Effect of using continuous integration (ci) and continuous delivery (cd) deployment in devops to reduce the gap between developer and operation," in 2022 International Arab Conference on Information Technology (ACIT). IEEE, 2022, pp. 1–8.