Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Analyzing the Ripple Effects of Refactoring

Mikel Robredo · Matteo Esposito · Fabio Palomba · Rafael Peñaloza · Valentina Lenarduzzi

Received: December 17, 2025/ Accepted: date

Abstract *Background.* Short development cycles and continuous delivery pressure often push developers toward expedients that lead to poor design and hard-to-maintain systems. A common remedy is code refactoring, which reduces complexity and improves maintainability, though often seen as costly and risky. *Aim.* We investigate the long-term effects of refactoring to provide recommendations that support strategic development decisions. *Method.* We assess refactoring impact through change- and defect-proneness analysis, as well as benefit/effort evaluation. *Results.* Most refactorings have short-lived effects, persisting for fewer than 10 changes. Structural refactorings may last over 190 changes, with significant differences across families. Medium-lived refactorings (9–19 changes) prove the most stable and efficient, while longer-lasting ones become increasingly defect-prone and costly. *Conclusions.* Refactorings differ in sustainability. Medium-duration refactorings strike the best balance between stability and maintenance cost, while structural ones, though impactful, pose higher long-term risks. These insights guide prioritization of refactoring types to maximize benefit and minimize technical debt.

Mikel Robredo
University of Oulu, Oulu, Finland
E-mail: mikel.robredomanero@oulu.fi

Matteo Esposito
University of Oulu, Oulu, Finland
E-mail: matteo.esposito@oulu.fi

Fabio Palomba
University of Salerno, Salerno, Italy
E-mail: fpalomba@unisa.it

Rafael Peñaloza
University of Milano-Bicocca, Milano, Italy
E-mail: rafael.penaloza@unimib.it

Valentina Lenarduzzi
University of Southern Denmark, Vejle, Denmark
University of Oulu, Oulu, Finland
E-mail: lenarduzzi@imada.sdu.dk

1 Introduction

In today's pervasive computing environment, where technology is integrated into our daily routines, keeping up with code maintenance poses an ever-expanding challenge (Esposito and Falessi, 2023). As development cycles shorten and the pressure to deliver quickly intensifies, developers may resort to quick-fix design solutions, ultimately leading to complex and unsustainable software systems (Lenarduzzi et al., 2021). Moreover, the involvement of non-software engineering developers in complex software projects is common, resulting in suboptimal design choices that further hinder maintenance. To mitigate this complexity, code refactoring is the classical solution (Kim et al., 2012). Refactoring aims to restructure code while preserving its external behavior, encompassing optimizations, architectural adjustments, and adopting design patterns (Kim et al., 2012; Lacerda et al., 2020). It has been the focus of the software engineering community for decades (Avgeriou et al., 2020; Esposito and Falessi, 2023; Lacerda et al., 2020; Lenarduzzi et al., 2021; Palomba et al., 2019; Peruma et al., 2022; Szóke et al., 2014).

By restructuring code, developers aim to enhance its design, readability, and maintainability, ensuring the long-term viability of software systems (Szóke et al., 2014). However, selecting the appropriate refactoring strategy can be daunting, prompting developers to seek guidance. Thus, there is a need for recommendations to assist developers in their refactoring efforts. Choosing the correct refactoring strategy can be challenging, leading developers to seek assistance. Thus, developers need recommendations to guide their refactoring effort (Peruma et al., 2022; Szóke et al., 2014). Nonetheless, developers perceive it as a significant time and resource allocation cost due to its complexity and limitations (Avgeriou et al., 2020; Kim et al., 2012). Hence, developers should be guided in optimizing refactoring efforts and their effect on the code base.

The state-of-the-art focus is on the relationship between refactoring and code quality, aiming at developing automatic or semi-automatic identification of developers' recommendations. For example, researchers focused on refactoring as a means of reducing software change and defect proneness (Mooij et al., 2020; Palomba et al., 2018), as well as refactoring side effects on non-functional aspects (Soares and et al., 2020) such as the introduction of code smells (Lacerda et al., 2020), code complexity (Sellitto et al., 2022; Soares and et al., 2020), or security vulnerabilities (Abid et al., 2020; Esposito et al., 2023; Iannone et al., 2023).

Although these studies have significantly advanced our understanding of refactoring, they share a key methodological limitation: they tend to evaluate the impact of refactorings over a **short time span**, typically analyzing changes between two consecutive commits or refactoring operations. This approach does not consider what we refer to as the *Ripple Effect* (RE): the **long-term propagation** of a refactoring's influence on the codebase, as reflected in subsequent maintenance activities, structural changes, or defect patterns. By neglecting this temporal dimension, previous research may underestimate the sustained or delayed consequences of refactoring decisions. For instance, a refactoring that initially appears beneficial, e.g., extracting a method or renaming a variable, might introduce structural dependencies or stylistic conventions that influence how developers modify the code in the future. These downstream effects, which may only surface after several subsequent changes, are not captured when analyses are confined to the immediate aftermath of a refactoring. As a result, current insights into the effectiveness and sustainability of refactoring remain limited, offering developers little guidance on the long-term implications of their design decisions.

In this paper, we propose a novel empirical investigation into the long-term RE of refactoring activities. We systematically analyze how different types of refactorings persist and influence the evolution of software systems over time. Our goal is to provide developers with actionable insights into which refactorings tend to produce stable, maintainable outcomes and which ones may incur hidden maintenance costs. To this end, we introduce a methodology that approximates the persistence of refactorings using *Bayesian Conditional Probability* (BCP) and *Longest Common Subsequence algorithm* (LCS).

Therefore, we analyzed the long-term effect of refactoring activities on software code via change and defective proneness of refactoring and benefit/effort ratio analysis testing. Therefore, our work makes the following contributions:

- We present the **first large-scale empirical study** analyzing the **long-term persistence of refactoring RE** across 98 refactoring types and 20 families, over 850k refactoring instances.
- We assess the **impact of RE on software maintainability**, highlighting its statistically significant effects on change-proneness and defect-proneness.
- We introduce and analyze a **change-efficiency metric** $\{CE\}$ to measure the benefit-to-effort trade-off of refactorings over time.
- We provide **practitioner guidelines** identifying when and what kind of refactoring operations are most sustainable and cost-effective in the long run.
- We publicly release all data and scripts to enable replicability, reproducibility, other than allowing researchers to build on top of our findings (See Section 9).

Our results show that most refactorings have short to medium-lived ripple effects, with **Add and Change** being the most frequent types. Structural refactorings exhibit **longer-lived but riskier** impacts, while localized changes fade faster but are more stable. Medium-duration refactorings (lasting 9–19 changes) emerge as a **sweet spot**, offering the **best trade-off between sustainability and cost-efficiency**. These findings inform both tooling and decision-making in long-term software maintenance.

Paper Structure. In Section 2, we provide the theoretical background for the study. Next, in Section 3, we present the study design and Section 4 the results. Subsequently, we discuss the obtained results in Section 5 while Section 6 presents our empirically-grounded guidelines for developers. Section 7 focuses on threats to the validity of our study. Section 8 discusses related work, and in Section 9, we draw the conclusions.

2 Background

In this section, we lay the theoretical groundwork for our study by describing the concept of the Ripple Effect (RE) and presenting the data analysis methods used to approximate its duration and impact.

2.1 Ripple Effect

The concept of RE stands for the propagation of an initial disturbance in a system, which may extend to further portions of the system through the initial impact and secondary disturbances generated indirectly Feynman (1963). Presented for the first time within the physics domain, this term has been previously adopted in diverse academic spheres such as Microeconomics Dolgui et al. (2018) and Sociology Barsade (2002), for instance. Haney (1972) first adopted the concept of RE within the Software Engineering domain to describe how a change in a module would trigger a change in another module. Yau et al. (1978) proposed a RE analysis technique as a complexity measure to evaluate and compare various program modifications within the scope of Software Maintenance. Subsequently, in 2001, Black proposed the Ripple Effect and Stability Tool (REST) tool, presented as an automated computation of the RE for C programs based on their approximation algorithm. To the best of our knowledge, this is the first empirical study investigating the adoption of RE of the code refactoring effect.

In our study, we adopt the RE term as the *long-term* propagation of the impact of a given refactoring R. To quantify the propagation, we probabilistically approximate the value of the impact of a refactoring on how stable the code introduced in a reference refactoring R_0 remains in subsequent refactorings. Therefore, the definition of RE as the impact propagation of a refactoring enables us to approximate for how long over time the code introduced in a given refactoring impacts the code base of a software system. Figure 1 depicts an example of the RE of a refactoring operation in a given class of a software system. The reference refactoring, i.e., R_0, or the initial refactoring operation that triggers the beginning of the RE stands at the beginning of the horizontal axis or refactoring activity over time (considering only refactoring operations of the same type performed in the same class). Since we defined the RE as the approximated probability of the share of lines introduced in R0 remaining unchanged over time, this probability is defined as $P(X) = 1$ at the stage R_0 is performed (vertical axis). Subsequently, the example depicts a probability update of the observed RE every time a refactoring operation of the same type is performed within the same affected class, that is, a probability update conditioned on subsequent refactoring operations (e.g. $P(X|R_1, R_2)$). For instance, a RE of probability $P(X|R_1, R_2) = 0.5$ would reflect that the shape of the code introduced in R_0 has been modified or removed by 0.5, and hence, it's RE impact remains at a probability of 0.5. Thus, this example aims to depict the RE propagation of R_0 as the conditional probability of its impact across time given subsequent refactorings. After R_0 is performed, a set of refactorings are performed subsequently by developers to improve the maintainability of the code R_1, R_2, R_3, ... , R_n where n depicts the refactoring in which we declare the end of R_0's RE. Given the novelty of the RE estimation approach employed in this study, we aimed at being as robust as possible when approximating the end of R_0's RE. Therefore, we defined that the impact of refactoring will no longer persist if the approximated probability of the R_0's remaining RE in the code base is $P(X) < 0.01(1\%)$.

The goal of this example, and therefore of the RE approach for understanding the long-term impact of a refactoring is to support developers on the refactoring choice, by providing approximated values on the duration of the impact of chosen refactoring. To this end, and given the conditional nature of the RE of refactoring operations on past observations, we approximate the probability of the persisting refactoring RE using the adopted techniques, BCP and LCS.

2.2 Bayesian Conditional Probability

Bayesian Conditional Probability (BCP) stands as a probabilistic approach for calculating the conditional probability of an event to happen, given how *likely* this event is to happen and given the knowledge of previous evidence (Hoijtink, 2009). Given the RE's definition within this study's context, the definition of the BCP could be adapted as the *approximated probability of the RE of a code refactoring remaining unchanged over time.* Through the approximated probability, this technique enables the estimation of future probabilities based on the updated evidence, i.e. new code refactorings of the same refactoring type given in the affected class. The BCP is mathematically defined through the Bayes' Theorem in Equation 1, which defines the conditional probability of an event A given the knowledge of an event B. This probabilistic approach provides the posterior probability of an event A happening given the likelihood of an event B happening if A has happened before, and therefore updates the prior knowledge (probability) of event A happening.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1}$$

The core mathematical strength of the Bayes' theorem relies on the independence of distributional assumptions, generally existing in most of the frequentist statistical techniques (Lee, 1997). Thus, this mathematical rule inverts the defined conditional probabilities, allowing one to find the probability of a cause given its effect (Gill, 2002). Within this model setting, the concerning events would be defined as follows:

- Event A: A line of code from the original refactoring R_0 remains unchanged.
- Event B: A subsequent refactoring R_i (or a series of them, e.g., R_1, R_2, ..., R_n) has been applied to the same affected class.

Similarly, we define the involved probabilities in the Bayes' theorem applied to the study's refactoring RE estimation as follows:

- $P(A)$: It is the initial (prior) probability that a line of code from the reference refactoring R_0 will remain unchanged in a subsequent refactoring. That is, this is the initial probability of the impact of the RE of the refactoring operation.
- $P(B)$: Represents the overall (marginal) probability that any line of code gets changed when a refactoring operation is performed.
- $P(B|A)$: Stands as the likelihood that a line of code gets changed in a subsequent refactoring, given that it was a line of code that had remained unchanged from the original reference refactoring R_0.
- $P(A|B)$: Represents the posterior probability that a line of code introduced in the reference refactoring R_0 remains unchanged after one or more subsequent refactorings of the same type (R_1, R_2, ..., R_n) have been performed within the same affected class.

Thus, BCP provides a probabilistic approximation of the posterior probability $P(A|B)$ (i.e. $P(X)$) for event A happening, i.e., probabilistically mirrors the development of the R_0's RE over time. Through its updating process, BCP provides continuous new evidence on the probability of event A happening after each new refactoring operation is performed, therefore updating future probability estimations. By adopting BCP, we aim to approximate the extent of the impact a given refactoring can have given the subsequent refactorings in the commit history of a software project. Thus, we define as RE the propagation of the BCP, and therefore the propagation of the impact of a refactoring, over time. We describe the probability estimation process in detail in Section 3.4.1.

2.3 Longest Common Subsequence

Longest Common Subsequence algorithm (LCS) is a commonly used technique for data comparison programs such as the *diff* utility (Hunt and MacIlroy, 1976) for instance. It has applications within multiple fields such as applied mathematics (Chowdhury and Ramachandran, 2006) or software engineering (Li et al., 2024) for instance. Given that we approximate the survival of the code introduced in R_0 as the *share of lines remaining unchanged in the analyzed class*, we adapt the LCS definition to approximate the probability of the impact of the initially introduced code in R_0 as follows:

$$P_i = \frac{2 \times |K_m(C_0, C_i)|}{|L_0| + |L_i|} \tag{2}$$

where:

- P_i (i.e. P(X)) is the share of lines remaining unchanged from C_0, presented as a similarity metric that can take a value between zero and one. $(0 \leq P_i \leq 1)$
- $K_m(C_0, C_i)$ is the number of matching code lines within the source code existing in C_0 and C_i.
- L_0 is the length of the longest common *substring* (sequence of lines of code for us) from the analyzed class source code at C_0.
- L_i is the length of the longest common substring (sequence of lines of code for us) from the analyzed class source code at C_i.

Thus, we approximate the RE generated by R_0 over time by quantifying the existing LCS in the affected Java class, and therefore the propagation of the impact of the concerning refactoring. We acknowledge that probabilistic techniques provide an approximation to reality, whatever the field of study might be. Therefore, we address this concern as a threat to validity in Section 7.

3 Empirical Study Design

This section describes the empirical study, including the study goal and research questions, the study context, the data collection methodology, and the data analysis approach. Our empirical study follows the established guidelines defined by Wohlin et al. (Wohlin et al., 2000). Figure 2 provides a graphical description of the study design. The numbers in the circle represent the RQs.

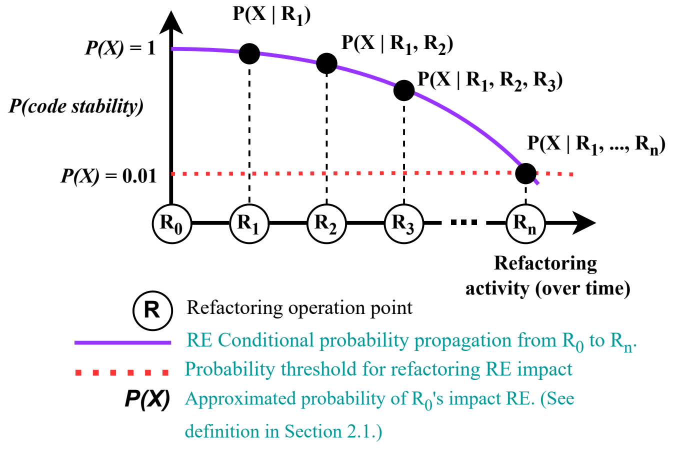
P(X | R1)
P(X | R1, R2)
P(X | R1, R2, R3)
P(X | R1, ..., Rn)
P(X) = 1
P(code stability)
P(X) = 0.01
R0
R1
R2
R3
Rn
Refactoring activity (over time)
R Refactoring operation point
RE Conditional probability propagation from R0 to Rn.
Probability threshold for refactoring RE impact
P(X) Approximated probability of R0's impact RE. (See definition in Section 2.1.)

Fig. 1: An example set of sequential refactorings to show the RE of the R_0.

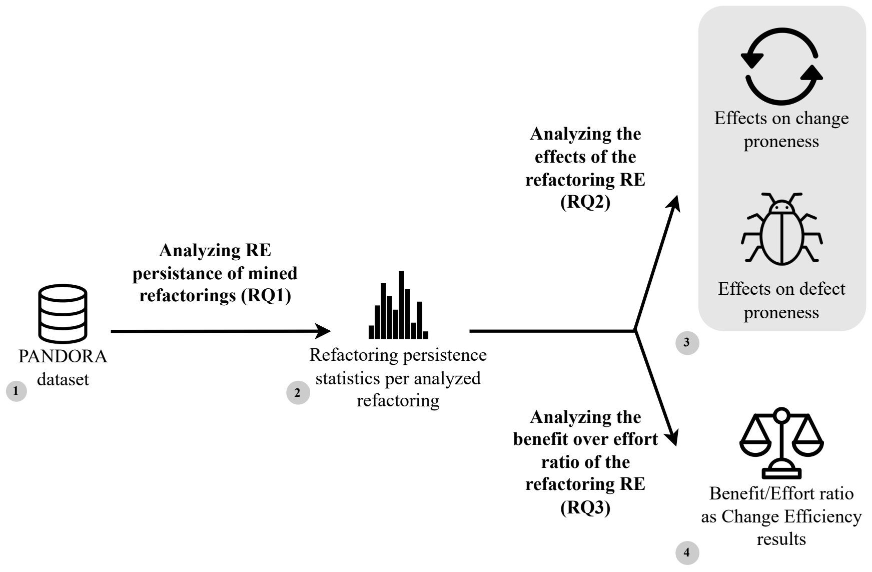
PANDORA dataset
1
Analyzing RE persistance of mined refactorings (RQ1)
Refactoring persistence statistics per analyzed refactoring
2
Analyzing the effects of the refactoring RE (RQ2)
Effects on change proneness
Effects on defect proneness
3
Analyzing the benefit over effort ratio of the refactoring RE (RQ3)
Benefit/Effort ratio as Change Efficiency results
4

Fig. 2: Study Workflow

3.1 Goal and Research Questions

The *goal* of this study is to analyze the long-term impact of refactoring activity on the code, with *purpose* to quantify the importance of performing refactoring during the development process. This will help developers recognize the benefits and efforts involved in refactoring their code and encourage them to undertake this process. The *perspective* of this study is to support developers and practitioners in measuring the relationship between benefit and effort when reflecting on refactoring. The study *context* is open source software (OSS) projects.

Therefore, we started developing the following first research question as follows:

RQ$_1$. *To what extent does the Ripple Effect (RE) of a refactoring activity persist in the code?*

We adopted the term RE from physics, which stands for the disturbance within a system that persists over time (Feynman, 1963). In more colloquial terms, this kind of disturbance resembles the impact of a stone when it drops into the water.

Cedrim et al. (2017a) conducted a study to assess the impact of a set of refactoring types on the durability of code smells over time. Their findings demonstrated that when employed for code smell deletion, specific refactorings can result in significant side effects. Similarly, Abid et al. (2022) highlighted the challenge of assessing the impact of refactoring when recommending developers to decide which refactoring type to use. Therefore, our *RQ* aimed to ascertain the longevity of the impact of refactoring on the code, analyzing its RE. To this end, we approximated the following computation of the probability of unchanged lines of code in subsequent refactorings. Therefore, to the best of our knowledge, this is the first study adopting the approximation of the RE to analyze the persistence over time of the code refactoring effect.

Furthermore, the impact of the refactorings performed in between is incorporated through the joint probability to capture the share of their change's impact on the persistence of the initial refactoring in the code. Moreover, the probability of long-term refactoring alone is not enough. We aimed to provide developers with recommendations on leveraging, thus benefiting, refactoring. Hence, we ask:

RQ$_2$. *What is the long-term effect of refactoring on change and defect proneness?*

In the context of this research question, we used the term *software entity* as an umbrella term for elements of source code such as packages, classes, methods, or lines of code. Developers continuously contribute to software projects by implementing or maintaining new features. Each software entity, therefore, is subject to change. The likelihood of changing a software entity is commonly referred to as change proneness (Arvanitou et al., 2017; Lu et al., 2012). A software defect is an unexpected computer program behavior that could go unnoticed (IEEE, 1990). In practical terms, fault proneness refers to the likelihood of identifying one or more faults within a software component (Denaro and Pezze, 2002). Software defects can be discovered through various verification and validation processes conducted at different points during development and maintenance (Denaro and Pezze, 2002; IEEE, 1990; Song et al., 2011). Refactoring is a behavior-preserving transformation of the source code (Abid et al., 2022). As such, albeit by preventing alteration in the behavior of software projects, it impacts both the development process and the quality of the product (Abid et al., 2022; Cedrim et al., 2017a; Sellitto et al., 2022). A refactor can deeply impact the current state of the source code; however, no prior study has ever analyzed the impact of a specific refactor in the long term.

Therefore, we aimed to analyze the change and defect proneness of refactoring over time to grasp its long-term effect on the code base.

Finally, the long-term effect of refactoring, whether beneficial or detrimental to the code base, does not happen spontaneously. Over time, developers put effort into making refactoring happen. Therefore, we ask:

RQ$_3$. *What is the benefit / effort ratio of long-term refactoring?*

Developers perceive that refactoring involves substantial cost and risks (Kim et al., 2012). Although refactorings are commonly associated with reducing software complexity, they can also negatively impact future inspection efforts (Kim et al., 2014). In practice, developers often do not confine refactoring to the rigorous definition of semantic-preserving code transformations (Kim et al., 2012, 2014). If, on the one hand, developers deliver a less complex software entity, on the other hand, they extend the lines of code, thus the future inspection effort(Kim et al., 2014). We aimed to provide developers with clear recommendations on long-term refactors; therefore, we computed their benefit / effort ratio to analyze whether the effort put into long-term refactoring is correctly awarded a beneficial effect on the software quality (see Section 3.3).

3.2 Study Context

As a context, we considered the PANDORA (Nguyen et al., 2022) dataset, which continuously collects data from December 2020, investigating the code quality in terms of code violations, technical debt, and code build stability among 365 Open Source Software (OSS). We selected the projects stored in the PANDORA dataset as they form part of the Apache Software Foundation (ASF).[1] This list of projects includes some of the most widely used software repositories, and can be similarly considered as mature industrial projects, mostly due to the strict review and inclusion process required by *The Apache Way*.[2] Moreover, the majority of the projects are implemented in Java, which is the language targeted by the refactoring mining tool we employ for analyzing refactoring activity in version control data (see Section 3.3).

Moreover, the data were also extended by Sellitto et al. (Sellitto et al., 2022), who analyzed the entire history of changes of the selected projects to determine which commits the developers applied to refactor.

Table 1 presents the summary statistics for the initially considered OSS projects. With a mean *age* of slightly more than 10 years, we can observe that even though the distribution of projects in terms of age is practically symmetric (mean and median are almost equal) the considered projects present high deviations in terms of Lines of Code and Commits, for instance. Such aforementioned distributional asymmetry can be clearly observed on the resulting skewness results. Consequently,

[1] `https://www.apache.org`
[2] `https://www.apache.org/theapacheway/`

Table 1: Summary descriptive statistics of the projects from the PANDORA dataset

	mean	sd	median	min	max	skew	kurtosis	se
Age	10.18	3.91	9.63	0.02	27	0.62	0.53	0.21
Size*	34,454.37	171,360.94	892.5	0	1,917,797	8.23	77.93	9,031.51
Stars	198.56	1,117.87	1	0	13,426	8.08	74.32	58.92
OI	11.88	85.93	0	0	920	9.18	84.16	4.53
CI	178.41	838.54	3	0	8,317	6.87	53.46	44.20
OvI	190.28	904.29	4	0	9,103	6.88	52.88	47.66
C	1,055.18	5,017.40	77	5	74,954	10.36	134.75	264.44
D	30.94	97.76	12	2	1,473	10.42	136.03	5.15
L	1.19	1.12	1	0	7	2.68	9.05	0.06
RT	26.20	28.16	16	0	102	1.08	0.08	1.48
RC	165.50	691.16	8	0	6,760	6.35	44.59	36.43
R	2,371.04	10,344.15	43	0	113,645	6.56	51.05	545.18

OI = Open issues, CI = Closed issues, OvI = Overall issues, C = Commits,
D = Developers, L = Programming Languages, RT = Refactoring Types
C = Refactoring Commits, R = Refactorings, (*) = Lines of Code

we consider the study context for this study to be representative for a varied topology of Java projects in terms of the presented software attributes. Additionally, we report that for the displayed **minimum** values presented for the attribute *size*, we only considered the Lines of Code (LOC) of languages recognized as programming languages based on the criteria adopted from the TIOBE index [3]. The TIOBE index requires three criteria from the included languages. First, a language must have an entry in Wikipedia that clearly states that it is a programming language. Secondly, it is *Turing complete*, and thirdly, it should have at least 5,000 hits for Google search +"<language> programming". The index contains 358 programming languages, which can be examined from the TIOBE documentation and a list provided in the replication package of this study. Therefore, and since some of the original projects in the PANDORA dataset had become obsolete modules (e.g. *apache/sling-org-apache-sling-starter-startup*), they reported minimum size values of 0.

Similarly, Figure 3 presents the histogram of the distribution of the creation year for the considered projects. We can observe a higher concentration of projects that were created during the last decade, which decreased towards the beginning of the twentieth decade, yet representing a higher population of recently created projects. Nevertheless, aiming at understanding the level of representativeness of the adopted projects, Figure 4 displays the distribution of some of the aforementioned software attributes based on the age of the projects through the *Ridge* plot [4]. The stacked Ridge plots displayed in Figure 4 describe the density of the considered projects in terms of different software attributes such as GitHub stars, size of the project (Lines of Code), issues and commits, and simultaneously presenting the results in different categories based on the age of the projects, thus allowing to understand the representativeness of the descriptive results observed in

[3] `https://www.tiobe.com/tiobe-index/programminglanguages_definition/`
[4] `https://r-graph-gallery.com/294-basic-ridgeline-plot.html`

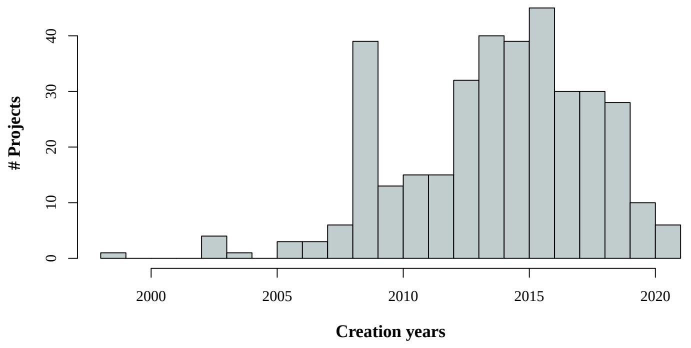

Fig. 3: Histogram of the distribution of the creation year for the considered 365 projects

Figure 3. From the displayed results, we can observe that projects reaching an age older than 21 years old result result to be the ones presenting the biggest project size, as well as the biggest number of registered issues, commits and GitHub stars. Looking at younger generation projects, the density values for the considered software attributes present lower values, yet visible gap between projects older and younger than 21 years from their creation clearly describes the high representativeness of older projects. Details from the considered 365 projects, as well as the scripts to generate the presented descriptive statistics are included in the shared replication package (see Section 9).

3.3 Data Collection

We report the data collection strategy we adopted in this study to gather the data needed to answer our RQs.

Collecting Refactoring Data. Since the study relied on refactoring operations performed during the development of the projects stored in PANDORA, and similarly, the dataset did not contain data on the refactoring activity, we only extracted the stored list of project names. In order to collect the refactoring activity performed in all the software repositories, we relied on the state-of-the-art refactoring mining tool RefactoringMiner (Alikhanifard and Tsantalis, 2025). RefactoringMiner can detect 103 refactoring types, through the analysis of how the Abstract Syntax Tree of a Java class/method has changed concerning one of the previous commits. The output of RefactoringMiner is formatted as a JSON file reporting for each commit the set of refactoring operations applied and the classes/methods subject to them. However, only a subset of traditional refactoring types is classified in the original Fowler catalogue (Fowler and Beck, 1999). Still, it can also identify types that cannot be mapped to the original catalog, such as composite refactoring *Move and Inline Method*. Finally, Table 2 shows the categorization of the 103 mineable refactoring types by RefactoringMiner, and

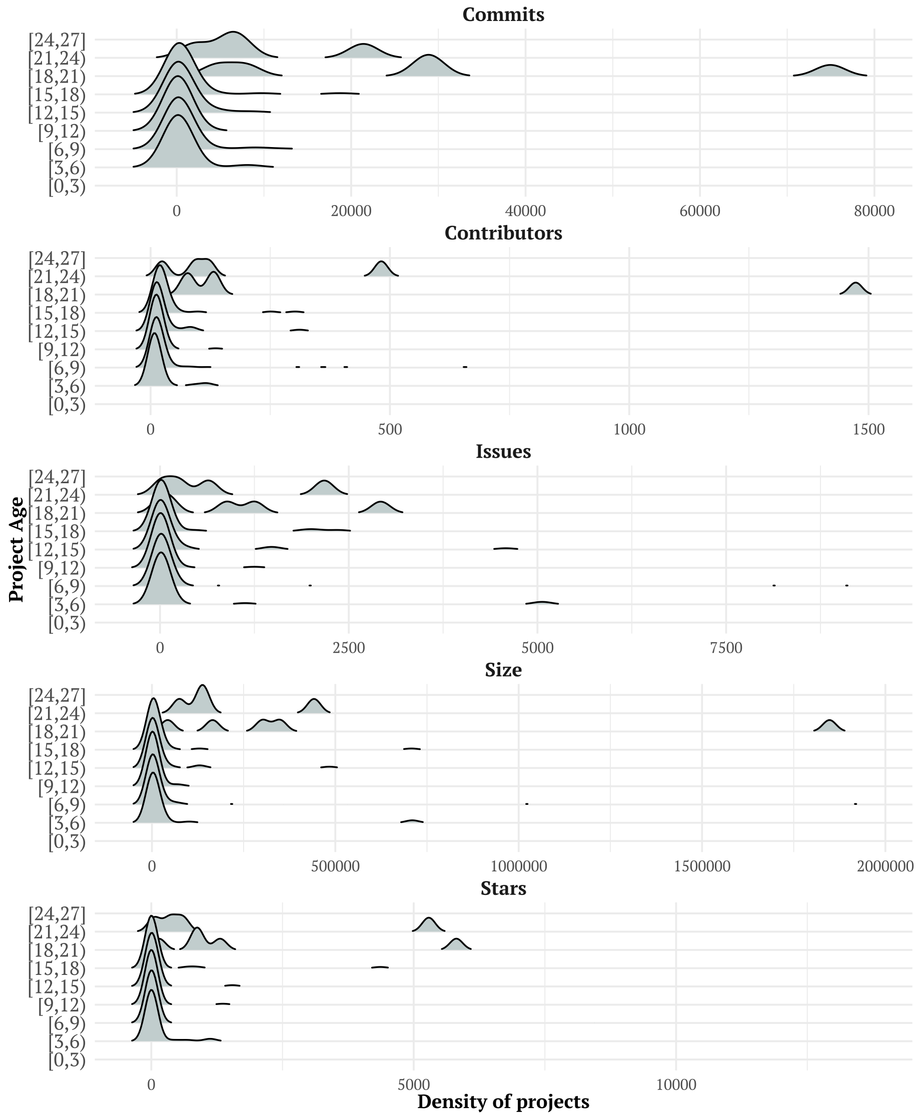

Fig. 4: Ridge plot with the density distribution of the considered projects for *Commits*, *Issues*, *Size* and *Stars* attributes.

hence, included in our study according to their type as defined by Fowler (1999). We are aware of other alternative refactoring detectors, for instance, RefDiff developed by Silva et al. (2020). However, RefactoringMiner has demonstrated the ability to reach a detection accuracy close to 100%, overcoming the capabilities of other existing tools Tsantalis et al. (2018), overcoming the capabilities of other existing tools. The performed adoption aims at reaching the best possible

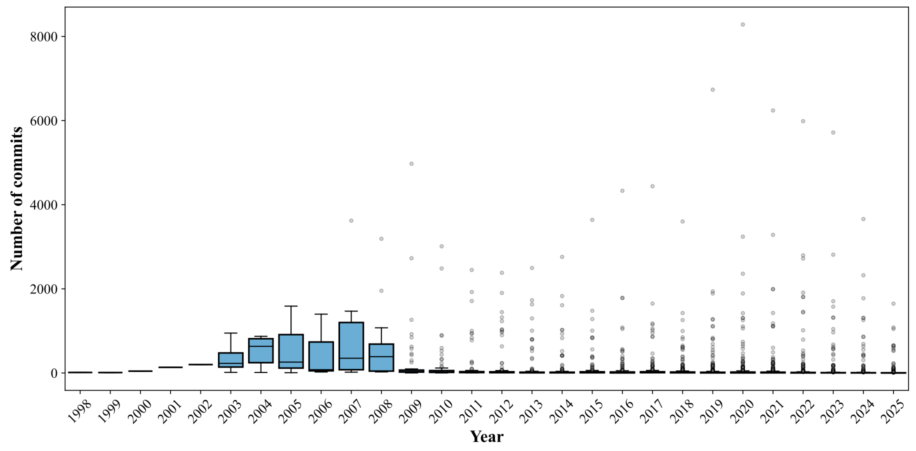

Fig. 5: Distribution on the commit activity of the studied projects across their life.

mining outcome by adopting the state-of-the-art refactoring mining tool. However, we acknowledge the possible corner cases in which the adopted tool might fail misclassifying spotted refactorings, as well as the limitation of refactoring analysis to Java code, leaving other programming languages aside from the study. We report this concern as a threat to validity in Section 7.

We started the data collection process, ensuring that all the projects listed in the PANDORA dataset remained active by the time the study was executed. For that, we implemented project exclusion criteria already used in existing literature. The adopted criteria consisted on first, excluding projects that reported being retired in GitHub (Amoroso d'Aragona et al., 2024). For that, we employed GitHub's REST API to extract the projects' metadata.[5] Subsequently, we cloned the remaining software repositories and mined their version control history during the last 12 months of life until the date we started with the data collection of our study. During this process, we excluded those projects that did not report a single commit during the considered period of time (Saarimäki et al., 2025). Moreover, we did not consider as accountable those commits made by bots (e.g. Github's dependa[bot]).[6] From the reported exclusion criteria no project reported to be archived in GitHub, but 20 projects were excluded from the total number of considered projects due to their inactivity during the last 12 months of life. Figure 5 displays the progression of the included projects' activity level distribution across their active life. Each boxplot represents the distribution in terms of commits of each of the projects that were active during that year.

Subsequently, we launched RefactoringMiner on the entire change history of the considered **365** projects to identify **88,863** commits where developers applied at least **853,573** refactorings. For that, we cloned the software repositories of the considered projects and launched RefactoringMiner on each of them. Thus, we mined the refactorings performed throughout their entire version control

[5] `https://docs.github.com/en/rest?apiVersion=2022-11-28`
[6] `https://github.com/dependabot`

Table 2: Refactorings Types provided by RefactoringMiner 3.0 and newer versions (Tsantalis et al., 2018)

Group	Description	Refactorings
Composing Methods	Re-organize how methods are composed, such as streamlining their logic or removing unneeded parts.	Extract / Inline / Merge / Split Method, Extract / Inline / Split / Merge / Rename Variable, Change Variable Type, Move Code (between methods), Merge Catch / Conditional, Split Conditional.
Moving Features between Objects	Re-organize the distribution of functionalities and data among classes.	Extract / Move / Rename Class, Move Method, Move Attribute, Localize / Reorder Parameter, Replace Attribute / Attribute with Variable.
Manage Objects Modifiers	Re-assign the modifiers from different parts of the code.	Change Attribute Access / Class Access Modifier, Change Type Declaration Kind, Add Method / Attribute / Variable / Parameter / Class Modifier, Remove Method / Attribute / Variable / Parameter / Class Modifier.
Organizing Data	Re-organize the way data is managed inside a class.	Extract / Split / Merge / Replace / Rename / Inline / Encapsulate / Parameterize Attribute, Change Attribute Type, Replace Variable With Attribute.
Simplifying Method Calls	Simplify class interactions by making the methods easier to call and understand.	Split / Merge / Add / Remove / Reorder / Rename Parameter, Parameterize Variable, Change Parameter Type, Change Method Access Modifier, Change Return Type, Rename Method.
Dealing with Generalization	Moving functionalities along class inheritance hierarchy.	Extract Superclass, Extract Subclass, Extract Interface, Pull Up / Push Down Attribute, Pull Up / Push Down Method, Split / Merge Class.
Object Replacement	Replace source code objects with alternatives.	Replace Loop With Pipeline / Anonymous With Lambda / Pipeline With Loop / Anonymous With Class / Generic With Diamond / Conditional With Ternary.
Package Management	Re-organize how packages are composed.	Rename / Move / Split / Merge Package.
Test Specific	Handle test specific scenarios.	Parameterize Test, Assert Throws / Timeout
Others	Other composite refactorings are detected by RefactoringMiner.	Move And Rename Attribute, Move And Inline Method, Move And Rename Class, Move And Rename Method, Extract And Move Method, Add / Modify / Remove (Class / Attribute / Method / Parameter / Variable) Annotation, Add / Change / Remove Thrown Exception Type, Change Package, Move Source Folder, Try With Resources, Invert Condition, Collapse Hierarchy.

history. Among the different approaches the adopted mining tool allows on how to be used, we installed and used the tool in the *command line*. From the mined outcome, we filtered out commits in which RefactoringMiner did not detect refactoring activity. Moreover, since we are interested in the temporal behavior of the refactoring RE, we reordered the mined commits with refactoring activity in chronological order since we found this not to exist in the raw mined data obtained from the adopted tool. We provide further details on the tool installation and suggestions from our experience mining the considered projects with this tool in the provided online appendix, which can be found in the replication package of this study (Section 9). From the resulting mined data, only **298** projects out from the initial 365 reported commits with refactoring activity, leaving **67** projects out from the data analysis due to non-existing data to analyze on them. Figure 6 provides a visual representation of the performed data collection process, which is later expanded with the data analysis process diagram in Figure 7.

Mining the Change History. To measure what is the long-term effect of refactoring, according to Palomba et al. (2018), we computed the proneness of the refactoring change ($\mathcal{C}$) and the defect ($\mathcal{D}$) of the refactoring. We computed C by extracting the change logs from the versioning systems of projects contained in the PANDORA data set (see Section 3.4.2). For that, we leveraged Python's PyDriller software repository mining library (Spadini et al., 2018), which can extract version control data such as the commit message, the number of developers, modifications, diffs, and the source code of a commit, among others, from cloned repositories. To compute D, we extracted data from the project management system and collected defect ticket information. Given that the totality of the projects stored in the adopted dataset belonged to the ASF, we mined their ticket information from GitHub and Jira platforms.[7][8] For that, we used the REST API endpoints facilitated by each of the project management systems. Subsequently, we computed C and D as follows:

$$\mathcal{C}(R_i, r_j) = \nu_{\mathcal{C}}(R_i)_{r_{j-1} \to r_j} \quad (3)$$

[7] `https://github.com`

[8] `https://www.atlassian.com/software/jira`

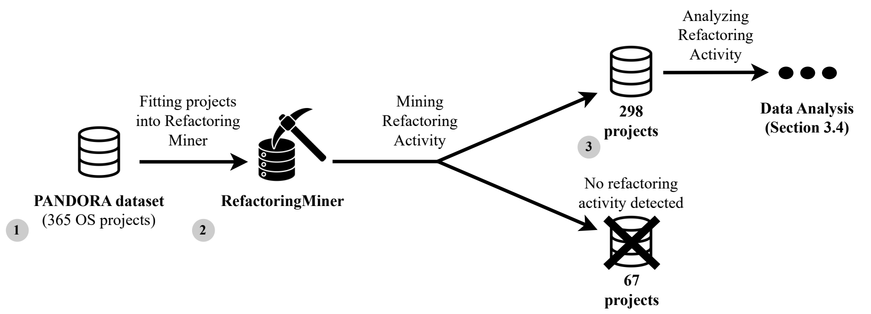
Fitting projects into Refactoring Miner
Mining Refactoring Activity
Analyzing Refactoring Activity
PANDORA dataset (365 OS projects)
1
RefactoringMiner
2
298 projects
3
No refactoring activity detected
67 projects
Data Analysis (Section 3.4)

Fig. 6: Data collection process diagram.

$$\mathcal{D}(R_i, r_j) = \nu_{\mathcal{D}}(R_i)_{r_{j-1} \to r_j} \tag{4}$$

Where $\nu_{\mathcal{C}}(R_i)_{r_{j-1} \to r_j}$ is the number of changes performed on the $i-th$ refactoring (R_i) by developers between the r_{j-1}'s and the r_j 's refactor dates. Similarly, $\nu_{\mathcal{D}}(R_i)_{r_{j-1} \to r_j}$ denotes number of bugs discovered on the $i-th$ refactoring (R_i) by developers between the r_{j-1}'s and the r_j 's refactor dates.

Collecting Developers Effort. We collected the total count of touched lines of code (TLOCs) for each refactoring and, therefore, for each developer that authored the refactoring. Considering the defined setting in which we track subsequent refactoring commits detected by RefactoringMiner, based on the fact that they have the same type as the reference refactoring R_0 and are applied in the same Java class, we computed the definition of TLOC from two different perspectives. As the first perspective, we considered as TLOC the added and deleted pure lines of code registered from each of the subsequent refactoring commits towards the reference refactoring R_0. Such collection of TLOC could be defined as follows:

$$\Delta TLOC_{r_0 \to r_i} = (ADD + DEL)_{r_0 \to r_i} \tag{5}$$

Where:

- $\Delta TLOC_{r_0 \to r_i}$ is the number of TLOC collected between the compared refactoring commits.
- ADD resembles the number of added lines of code.
- DEL resembles the number of deleted lines of code.
- r_0 represents the source code at the stage of the reference refactoring commit.
- r_i represents the source code at the stage of the i^{th} subsequent refactoring commit collected.

Similarly, and aiming to detect the specific effort performed not between refactoring commits but between the current refactoring commit and its direct predecessor, we also computed TLOC based on the added and deleted pure lines of code registered from the current refactoring commit towards its parent commit, as follows:

$$\Delta TLOC_{r_{i-1} \to r_i} = (ADD + DEL)_{r_{i-1} \to r_i} \tag{6}$$

Where considering the new definitions:

- $\Delta TLOC_{r_{i-1} \to r_i}$ is the number of TLOC collected between the analyzed subsequent refactoring commit and its parent commit.
- ADD resembles the number of added lines of code.
- DEL resembles the number of deleted lines of code.
- r_{i-1} represents the source code at the stage of the subsequent refactoring commit's parent commit.
- r_i represents the source code at the stage of the i^{th} subsequent refactoring commit collected.

We used PyDriller to mine the number of added and deleted lines of code and, therefore, to perform the computations described in each of the commits with refactoring activity. We adopted TLOC as a proxy for measuring developer's effort, grounded in prior empirical software engineering work (Çarka et al., 2022; Lavazza et al., 2025). While this metric captures syntactic change volume, it may not capture more semantic or cognitive effort metrics, for instance, code churn complexity or developer experience. We acknowledge this constraint as a validiy threat in Section 7.

Mining the Bugs Fixing Commits. In this stage, we considered the issue ticket information mined in previous stages of the data collection from GitHub and Jira issue tracking systems. We identified bug fix commits employing regular expressions to check the presence of IDs in the versioning system change log, e.g., "fixed issue #ID" or "issue ID". For each issue ID related to a commit, we downloaded the corresponding issue reports from their issue tracking system and extracted the following information from them:

- Project name
- Issue type (i.e., whether an issue is a bug, enhancement request, etc.)
- Status of the issue (i.e., whether an issue was closed or not)
- Resolution of the issue (i.e., whether an issue was resolved by fixing it, or whether it was a duplicate bug report or a case "works for me" case)
- Issue opening date
- Issue closing date (if available)

As we report in Section 4, we found issue reports from the analyzed projects only in GitHub and JIRA Issue Tracking Systems (ITS). To collect the issue report data from the mentioned ITS, we leveraged GitHub's REST API[9] as well as JIRA's REST API v3.[10] Likewise, we leveraged the already-mentioned *PyDriller* again (Spadini et al., 2018) Python package to mine the messages committed within the analyzed refactoring commits. Based on the structure of the tickets used in each of the mined ITS, we leveraged the following regular expressions:

- *JIRA issue ticket pattern:* Matches issue tickets commonly used in JIRA, where a project identifier is followed by the issue number (e.g. `ANY23-588`, `XMLBEANS-655`):

  ```
  [A-Z]{2,}-\d+
  ```

- *GitHub issue ticket pattern:* Matches issue tickets commonly used in GitHub, where the issue ticket number is preceded by a # symbol (e.g., `#296` fetched from *Apache Gora* project):

  ```
  #\d+
  ```

- *Alternative numeric issue ticket pattern:* Matches issue tickets commonly referenced in commit messages, where only the raw issue ticket number is provided (e.g., `17073` fetched from *Apache DolphinScheduler* project):

  ```
  \d{4,}
  ```

[9] https://docs.github.com/en/rest?apiVersion=2022-11-28
[10] https://developer.atlassian.com/cloud/jira/platform/rest/v3/intro/#about

This approach is the best effort, but it might not capture the totality of bug-fixing commits given the existing heterogeneity when referencing issue tickets in commit messages. We report this evidence as a threat to validity in Section 7. We report this evidence as a threat to validity in Section 7. Similarly, in order to quantitatively assess the potential threat of using the adopted pattern recognition approach, and thus mitigate the impact of possible false positives in bug fixing commits, we performed a manual validation experiment to assess the validity of our results. We describe the design of the experiment, as well as the results in Appendix A.

Change Efficiency as a "Benefit/Effort Ratio". Existing literature has measured the benefit of code changes in different ways. Ó Cinnéide et al. (2016) highlighted the potential use of code smells reduction and the improvement on software quality metrics as a way to measure the impact of refactoring. Subsequent works introduced further measurement techniques positively valued by industry practitioners, such as technical debt reduction (Tempero et al., 2017) as well as test smell reduction (Soares et al., 2022). Moreover, in a recent study implemented by Lima et al. (2023), industry practitioners claimed to interpret the benefits of code changes in terms of an increase of code reusability, as well as the improvement of the code's readability level.

In our study, we defined Change Efficiency ($\mathcal{CE}$) as the ratio of the number of changes $\nu_{\mathcal{C}}(R_i)_{r_{j-1}\rightarrow r_j}$ to the number of Effective Lines of Code (ELOC) $ELOC(R_i, s_j)$. We defined this metric as a new interpretation of the commonly known change-proneness metric (Lenarduzzi et al., 2020; Palomba et al., 2018) employed within inter-refactoring commit analysis. This measures the degree of change per line of code between two successive refactoring. It is important to note that CE captures syntactic effort, and therefore, should not be interpreted as a measure of cognitive refactoring effort.

$$\mathcal{CE}(R_i, r_j) = \frac{\nu_{\mathcal{C}}(R_i)_{r_{j-1}\rightarrow r_j}}{ELOC(R_i, r_j)} \tag{7}$$

Where $\nu_{\mathcal{C}}(R_i)_{r_{j-1}\rightarrow r_j}$ represents the number of changes that developers added to the $i-th$ refactoring (R_i) from the r_{j-1} to the r_j 's refactoring dates. These can be bug fixes, feature additions, and other changes to make the software more functional, faster, or easier to maintain. Each change theoretically adds value to the software by improving current features, eliminating defects, or mapping the product into new requirements. Since we are interested in the RE of the impact introduced by the reference refactoring R_0, we computed $\mathcal{CE}$ across all the collected refactorings performed within the originally affected Java class. Similarly, aiming at providing consistency to the significance of the computed $\mathcal{CE}$ as a refactoring benefit measurement, we only considered as the consecution of the RE those refactoring operations of the same type as the references refactoring.

$ELOC(R_i, r_j)$ is the number of effective lines of code in refactoring R_i between refactoring date r_{j-1} and r_j. We adhere to the definition of an effective line of code as a non-empty line not containing only a curly bracket or starting with "://", "/*", or "*" from Lenarduzzi et al. (2020). ELOC measures the quantity of touched code in the refactoring activity. A higher ELOC generally suggests more extensive changes to the software entity. We adopted the SCC command-line tool to extract this metric from the mined repositories at each stage of their

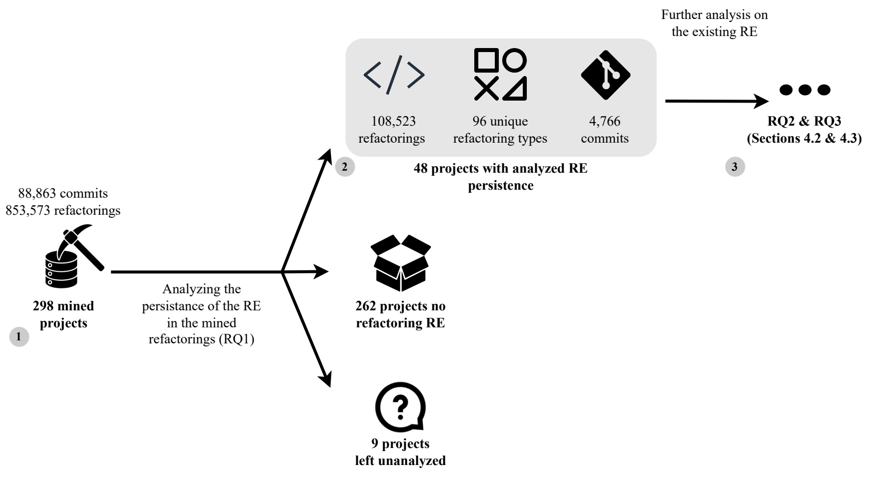
Further analysis on the existing RE
108,523 refactorings
96 unique refactoring types
4,766 commits
48 projects with analyzed RE persistence
RQ2 & RQ3 (Sections 4.2 & 4.3)
88,863 commits
853,573 refactorings
298 mined projects
Analyzing the persistance of the RE in the mined refactorings (RQ1)
262 projects no refactoring RE
9 projects left unanalyzed
1
2
3

Fig. 7: Ripple Effect persistence analysis process diagram.

development process when refactoring activity was detected.[11] Previously used in the literature (Saarimäki et al., 2025), SCC is a tool designed for calculating the lines of code, blank lines, comment lines, and physical lines of source code in many programming languages, among other metrics.

Higher $\mathcal{CE}$ values mean that much change (**benefit**) is being achieved relative to the code authored (**effort**). It can mirror agile, responsive development practices where changes are well integrated into the system. It can hint at a healthy development environment where the team can respond rapidly to new issues or requirements. Conversely, lower $\mathcal{CE}$ values can indicate a mature, stable product that requires fewer changes or a potential underutilization of development effort. To this extent, the ratio helps to identify how effectively the development effort (as measured by ELOC) can yield tangible improvements or necessary changes in the software. The metric also balances the code quality and the amount of changes. High-quality code might require less change (Esposito and Falessi, 2023), leading to a lower $\mathcal{CE}$ that would be preferred in environments where stability and precision matter. In environments where innovation and rapid iteration matter, a higher $\mathcal{CE}$ would be better.

All the data collection stages were conducted on a workstation equipped with an Intel Core i9-13900KF CPU (24 cores, 32 threads, base frequency 3.00 GHz), with 64 GB of RAM and 7.27 TB of available storage.

3.4 Data Analysis

This section describes the data analysis process and the techniques and methods for answering our RQs. Figure 7 depicts the procedure undergone to follow the RE persistence on the analyzed refactorings.

[11] `https://github.com/boyter/scc/tree/master`

3.4.1 Analyzing the persistence of the RE of refactoring in code (RQ_1)

We analyzed the RE of the impact of refactoring on the code over time. As explained previously, we defined RE as the length of time that the impact of the code introduced in a refactoring remains in the code base. To capture the impact subsequent refactorings have on the RE of the impact of the initial refactoring as well, we considered two approaches: *Bayesian Conditional Probability* (Hoijtink, 2009) and, as an alternative approach to quantifying the persistence of the RE of refactoring in code, *Longest Common Subsequence* (Chvatal and Sankoff, 1975). As in Section 2, we considered the RE of the impact of the code introduced in R_0 to be over when its probability of remaining in the code base is $P(X) < 0.01$ (1%).

Bayesian Conditional Probability (BCP): We calculated the BCP of the *share of lines remaining unchanged* from the reference refactoring R_0 after performing each subsequent refactoring as a statistical approximation to the RE of the impact R_0 could have over time. For this, and since the number of lines modified between R_0 and R_1 can be obtained through the RefactoringMiner tool, we calculated the *prior* probability of the share of lines remaining unchanged. This allows us to approximate the probability of the RE of R_0 in R_n as the *posterior* conditional probability, where n would resemble the last refactoring in which the RE of the R_0 would endure.

Considering Bayes' theorem defined in Equation 1 as the framework to describe our probabilities, we defined them as follows:

- ***Prior probability*** *(P(A)):* Probability that a line of code from R_0 remains unchanged in the subsequent refactoring.
- ***Marginal probability*** *(P(B)):* Overall probability that a line gets changed after performing refactoring activity.
- ***Likelihood*** *(P(B|A)):* Probability that a line of code introduced in R_0 gets changed in a subsequent refactoring.
- ***Posterior probability*** *(P(A|B)) = P(X)):* Probability that a line of code introduced in R_0 remains unchanged after subsequent refactoring.

As explained in Section 2, *Bayesian theory* emphasizes the continuous updating of the hypothesis probability. Consequently, our initial *prior knowledge* is based on the average change in lines of code observed in all the collected refactorings. Similarly, in each of the analyzed refactorings, the *likelihood* is founded on the binomial distribution of the lines changed in the first refactoring R_0, as demonstrated in Equation 8. Here, n is the total number of lines before the new refactoring, k is the number of changed lines, and p is *prior.*

$$L = \binom{n}{k} \times p^k \times (1-p)^{n-k} \tag{8}$$

Once the *prior* and *likelihood* (L) have been calculated, we greedily calculated the *posterior* probability until the defined threshold is met. This approximates the RE of the impact of the reference refactoring, R_0. The *Bayesian updating* approach enables the analysis of the RE of the impact of different types of refactorings, as well as the approximation on the impact of the definition of *inter-refactoring* activity performed from R_0 until R_n.

The proposed method enabled us to calculate the length of the RE and the impact the analyzed refactorings have over time, and therefore explore the RE of various types of collected refactorings.

Longest Common Subsequence (LCS): This approach provides a *line-by-line* similarity comparison between the analyzed versions since we aimed to remain as close as possible to the initial definition presented in the original approach. Thus, it calculates the similarity among subsequent class versions affected by the same refactoring type between R_0 and R_n. However, we added as threats to validity (see Section 7) possible downsides for adjusting to the initial problem definition.

To approximate the survival of the code introduced in R_0 as the *share of lines remaining unchanged in the analyzed class*, we defined the approximation towards the probability on the impact of the initially introduced code in R_0 as follows:

$$P_i = \frac{2 \times |K_m(C_0, C_i)|}{|L_0| + |L_i|} \tag{9}$$

where:

- P_i (i.e. P(X)) is the **share of lines remaining unchanged** from C_0, presented as a similarity metric that can take a value between zero and one. $(0 \leq P_i \leq 1)$
- $K_m(C_0, C_i)$ is the number of **matching code lines** within the source code existing in C_0 and C_i.
- L_0 is the **length of the longest common *substring*** (sequence of lines of code for us) from the analyzed class source code at C_0.
- L_i is the **length of the longest common *substring*** (sequence of lines of code for us) from the analyzed class source code at C_i.

Similarly, as in the BCP approach, we implemented LCS in a continuous update setting on the hypothesis probability. Therefore, the initial probability (P_0) consists of the initial probability of the share of lines of code remaining unchanged in the analyzed class (i.e. equals to 1). Subsequently, the approximated probability will be updated based on the changes implemented in the analyzed class through the same type of code refactoring. This approximates the RE of the impact of the reference refactoring R_0 in subsequent commits where R_i of the same type are applied to the analyzed class up until the R_n refactoring, where n would resemble the stage in the class where the impact of R_0 is less than the defined threshold. Alternatively, R_n could also be the last refactoring of the same type found to affect the same class during the entire version control history of the class, even if the RE would still be found to last in its source code. The LCS approach thus acts as an alternative, yet valid approach to quantifying the persistence of the RE of refactoring in code.

Hypothesis verification: To support our analysis, we conjectured three hypotheses as follows:

Different refactoring families may involve distinct levels of complexity and scope, which could impact the time required for RE to propagate. However, without formal testing, it remains unclear whether the observed differences in RE duration are systematic or simply due to random variation. By testing this null hypothesis, we assess whether the family-level classification of refactorings actually captures meaningful differences in duration.

- H_{01}: *There is a statistically significant difference in the duration of RE across all the refactoring families.*

– H_{11}: *There is no difference in the duration of RE across all the refactoring families.*

While H_1 looks at differences across all families simultaneously, H_2 focuses on pairwise comparisons between families. This distinction allows us to detect whether particular families are especially different in their temporal behavior, even if the overall distribution across families does not differ significantly. Testing this hypothesis is important because developers may allocate resources differently depending on which types of refactorings are expected to cause longer or shorter ripple effects. Hence, we define H_2 as follows:

– H_{02}: *There is a statistically significant difference in the duration of RE between refactoring family pairs.*
– H_{12}: *There is no difference in the duration of RE between refactoring family pairs.*

Similarly, and since we are employing two techniques to estimate the duration of the RE of refactoring, we define a third hypothesis as follows:

– H_{03}: *There is a statistically significant difference in the observations of Bayesian or LCS over time.*
– H_{13}: *There is no difference in the observations of Bayesian or LCS over time.*

To validate our hypothesis, we tested for the normality of the data beforehand. Therefore we conjectured two **hypothesis** ($H_{\mathcal{N}}$) as follows:

– $H_{1\mathcal{N}1}$: *RE durations are not normally distributed.*
– $H_{1\mathcal{N}2}$: *Bayesian or LCS observations are not normally distributed.*

Hence, we defined the **null hypothesis** (H_0) as follows:

– $H_{0\mathcal{N}1}$: *RE durations are normally distributed.*
– $H_{0\mathcal{N}1}$: *Bayesian or LCS observations are normally distributed.*

The PANDORA dataset contains data on the 365 OSS project. Hence, we tested $H_{\mathcal{N}}$ hypotheses with the Anderson-Darling (AD) test (Anderson and Darling, 1952). The AD test assesses whether data samples derive from a specific probability distribution, such as the normal distribution. AD measures the difference between the sample data and the expected values from the tested distribution. More specifically, it evaluates differences in the cumulative distribution function (CDF) between the observed data and the hypothesized distribution (Anderson and Darling, 1952).

The AD and the Shapiro-Wilk (SW) (Shaphiro and Wilk, 1965) are both statistical tests used to assess the normality of data. The SW test focuses on the correlation between the observed data and the expected values under a normal distribution, emphasizing the smallest and largest values in the dataset. Therefore, according to Mishra et al. (2019), SW is more appropriate when targeting small datasets (≤ 50 samples). On the other hand, the AD test considers a wider range of values, including those in the middle of the distribution, providing a more sensitive evaluation of normality, especially for larger sample sizes than our own. Therefore, we preferred AD over SW (Stephens, 1974). AD is also considered one of the most powerful statistical tools for detecting most departures from normality (Stephens, 1974, 2017).

In case of **acceptance of the null hypotheses ($H_{\mathcal{N}}$)**, similarly to the normality testing sample concerns, we will test H_{11} and H_{13} with the Z-statistic (Sprinthall, 2011). We used the Z-statistic to assess the significance of observed differences between sample data and population parameters. Calculated by standardizing sample statistics based on known population parameters, the Z-statistic quantifies the extent to which observed results deviate from what would be expected under the null hypothesis (Casella and Berger, 2002; Montgomery and Runger, 2020; Sprinthall, 2011).

Interpreting the Z-statistic involves comparing its value with critical values of the standard normal distribution, typically determined by the chosen significance level (Casella and Berger, 2002). If the calculated Z statistic falls within the critical region, it suggests that the observed results are extreme enough to reject the null hypothesis in favor of the alternative (Montgomery and Runger, 2020).

In case of **rejection of the null hypothesis ($H_{\mathcal{N}}$)**, we tested H_{11} and H_{13} using the Wilcoxon signed rank test (WT) (Wilcoxon, 1945), which is a nonparametric statistical test that compares two related samples or paired data. WT uses the absolute difference between the two observations to classify and then compare the sum of the positive and negative differences. The test statistic is the lowest of both. We selected WT to test H_{11} and H_{13} because LCS and RE observations would not be normally distributed; hence, we used it instead of the paired t-test, which assumes a normal data distribution. We will test H_2 using Dunn's all-pairs test (Dunn, 1961). The Dunn All-Pairs Test, also known as the Dunn Test (**DT**) or Dunn-Bonferroni Test, is a post hoc test that compares the differences between all group pairs in a dataset. DT compares the rank sums of all possible group pairs and obtains a p-value set for each paired comparison.

We set our alpha to 0.01. We decided to reduce the alpha value from the standard value of 0.05 due to the many statistical tests we performed and therefore to reach a good balance between Type I and II error (Kim and Choi, 2021). To the best of our knowledge, there is no previous study introducing a probabilistic threshold for measuring the longevity of refactoring's RE. Therefore, to assess the robustness of this choice, we conducted a sensitivity analysis by applying alternative thresholds and inspecting the difference in the number of observed REs, as well as their average RE longevity. We present the results from the sensitivity analysis in Appendix C.

3.4.2 Analyzing the long-term effect of refactoring (RQ_2)

To measure what is the long-term effect of refactoring effect, we computed the proneness of the refactoring change ($\mathcal{C}$) and the defect ($\mathcal{D}$) of the refactoring as described in Section 3.3.

Therefore, to answer RQ_2, we identified the **dependent variable**, $\mathcal{C}$ and $\mathcal{D}$, and the **independent variable**, time. Both $\mathcal{C}$ and $\mathcal{D}$ are **numerical continuous variables**. Hence, we analyzed the trend over time of $\mathcal{C}$ and $\mathcal{D}$. To support our analysis we conjectured one **hypothesis** (H_1) as follows:

- H_{04}: *There is no difference in the observations of $\mathcal{C}$ over time.*
- H_{05}: *There is no difference in the observations of $\mathcal{D}$ over time.*
- $H_{1\rho 1}$: *There is a no correlation between $\mathcal{C}$ observations and the RE duration.*
- $H_{1\rho 2}$: *There is a no correlation between $\mathcal{C}$ observations and the RE duration.*

Hence, we defined the **null hypothesis** (H_0) as follows:

- H_{14}: *There is a statistically significant difference in the observations of $\mathcal{C}$ over time.*
- H_{15}: *There is a statistically significant difference in the observations of $\mathcal{D}$ over time.*
- $H_{1\rho 1}$: *There is a statistically significant correlation between $\mathcal{C}$ observations and the RE duration.*
- $H_{1\rho 2}$: *There is a statistically significant correlation between $\mathcal{C}$ observations and the RE duration.*

To test our hypothesis, we will test for the normality of the data beforehand. Therefore we conjectured two **hypothesis** ($H_{\mathcal{N}}$) as follows:

- $H_{1\mathcal{N}3}$: *$\mathcal{C}$ observations are not normally distributed.*
- $H_{1\mathcal{N}4}$: *$\mathcal{D}$ observations are not normally distributed.*

Hence, we defined the **null hypothesis** (H_0) as follows:

- $H_{0\mathcal{N}3}$: *$\mathcal{C}$ observations are normally distributed.*
- $H_{0\mathcal{N}4}$: *$\mathcal{D}$ observations are normally distributed.*

Similarly to RQ$_1$ tested $H_{1\mathcal{N}3}$ and $H_{1\mathcal{N}4}$ with AD. Depending on the rejection of the normality hypothesis we tested H_{04} and H_{05} with the Z-test or WT. Moreover based on the rejection of $H_{0\mathcal{N}3}$ and $H_{0\mathcal{N}4}$ we considered using parametric tests such as *Pearson's r* correlation coefficient (Cohen et al., 2009) or non-parametric tests such as *Spearman's ρ* or *Kendall's τ* (Wohlin et al., 2000) coefficient for instance.

3.4.3 Analyzing the ratio benefit / effort of refactoring (RQ3)

To answer RQ_3, we identified the **dependent variable**, benefit / effort ratio, and the **independent variable**, time. The benefit / effort ratio is a **numerical continuous variable**. Hence, we collected the developer's refactoring effort to grasp whether the developer's effort is rewarded with long-term benefits to the code base. Therefore, to support our analysis of the relationships, we conjectured a hypothesis (H_2) as follows:

- H_{16}: *There is no difference in the benefit / effort ratio over time.*

Hence, we defined the **null hypothesis** (H_0) as follows:

- H_{06}: *There is a statistically significant difference in the benefit / effort ratio over time.*

We tested the normality of the data with the following **hypothesis** ($H_{\mathcal{N}}$) as follows:

- $H_{1\mathcal{N}5}$: *the benefit / effort ratio observations are not normally distributed.*

Hence, we defined the **null hypothesis** (H_0) as follows:

- $H_{0\mathcal{N}5}$: *the benefit / effort ratio observations are normally distributed.*

Based on the acceptance or rejection of the null hypothesis ($H_{\mathcal{N}}$), we tested H_2 with the Z-test or WT.

4 Analysis of the Results

This section presents the findings of our study used to address each RQ.

4.1 Analyzing the persistence of the RE of refactoring in code (RQ_1)

In this section, we present the obtained results on the distribution of refactoring types concerning the RE, its extent, and our response to RQ_1.

We analyzed **108,523** instances of **98 unique refactoring types** (among the 103 detectable by RefactoringMiner[12] and reported in Table 2) in **48 projects** (among the 365 projects in PANDORA) and in **4766 commits** (Table 3). Extending the insights described on the data collection stage in Section 3.3 and following the strategy defined to analyze the survival of the impact of the RE of a reference refactoring R_0, described in Section 3.4, we tracked the version control history for each of the **853,573** mined refactorings. During the process, only the refactoring cases that would have future subsequent refactorings of *the same type* and within *the same affected Java class* would be considered for the analysis. Therefore, from the initially considered group of **298** projects containing mined refactorings, **262** were entirely processed without identifying a single refactoring case matching the defined analysis criterion. Similarly, **9** projects were not analysed due to the lack of time for processing them analyzing the version control history data for each of the existing Java classes affected by refactoring activity during the lifetime of the considered ASF projects resulted in a long computational process. We acknowledge this limitation as threat to validity, and therefore discuss about it in Section 7. Consequently, this study concluded the analysis for a total of **48** projects among the initially considered ones.

Among such instances, Change Variable Type (6.89%), Extract Method (5.16%), and Add Parameter (4.97%) are the most common refactoring types. The frequency of the refactorings, on average, is very low, with most types being just above the 1% threshold. The least frequently applied refactorings, Split Package and Split Attribute are only used 4 times, suggesting they are used only in specific contexts, thus highlighting strong bias toward using refactorings that make methods more readable and maintainable compared to more structural ones.

Due to the many refactoring types, we grouped the 98 refactorings into 20 unique families, as shown in Table 4. The most common refactoring operations belong to the "**Add**" family (21.74%) and "**Change**" family (20.48%), which totals 40% of all the refactoring in our collected data. Hence, this shows that additive and modification refactoring are the operations that are performed most often compared to structural and organizational ones. RF aggregation allows for a more structured and interpretable analysis by reducing redundancy and highlighting broader refactoring trends. Moreover, focusing on refactoring families instead of individual refactorings makes the comparison more explicit, mitigates the impact of highly specific or infrequent refactorings, and enables a more meaningful discussion of RE patterns. Therefore, we structured our discussion around this **20 refactoring families** (RF) for the remainder of our work.

[12] `https://github.com/tsantalis/RefactoringMiner`

Table 3: Distribution of Refactorings by Type (RQ_1)

Refactoring Type	#	%	Refactoring Type	#	%
Change Variable Type	7478	6.89	Replace Variable With Attribute	366	0.34
Extract Method	5597	5.16	Change Class Access Modifier	310	0.29
Add Parameter	5396	4.97	Assert Throws	287	0.26
Rename Method	4348	4.01	Move And Rename Class	286	0.26
Change Parameter Type	4308	3.97	Invert Condition	265	0.24
Rename Variable	4186	3.86	Modify Parameter Annotation	248	0.23
Add Method Annotation	4050	3.73	Move Code	246	0.23
Rename Parameter	3863	3.56	Extract Class	230	0.21
Remove Attribute Modifier	3661	3.37	Remove Attribute Annotation	218	0.20
Add Class Annotation	3659	3.37	Localize Parameter	192	0.18
Change Return Type	3587	3.31	Add Class Modifier	188	0.17
Change Method Access Modifier	3493	3.22	Replace Attribute With Variable	177	0.16
Move Method	3257	3.00	Move And Rename Method	148	0.14
Add Variable Modifier	3205	2.95	Try With Resources	139	0.13
Extract Variable	3077	2.84	Split Conditional	134	0.12
Remove Thrown Exception Type	2560	2.36	Push Down Method	127	0.12
Modify Method Annotation	2510	2.31	Merge Parameter	118	0.11
Add Parameter Modifier	2340	2.16	Reorder Parameter	110	0.10
Remove Parameter	2316	2.13	Remove Variable Annotation	107	0.10
Extract And Move Method	2176	2.01	Replace Conditional With Ternary	107	0.10
Move Class	1964	1.81	Extract Superclass	98	0.09
Remove Class Annotation	1826	1.68	Move And Inline Method	95	0.09
Remove Method Annotation	1799	1.66	Parameterize Attribute	91	0.08
Rename Attribute	1676	1.54	Remove Class Modifier	91	0.08
Add Parameter Annotation	1659	1.53	Replace Loop With Pipeline	91	0.08
Change Attribute Type	1574	1.45	Move Source Folder	90	0.08
Inline Variable	1312	1.21	Split Method	79	0.07
Parameterize Variable	1196	1.10	Replace Anonymous With Lambda	63	0.06
Replace Generic With Diamond	1143	1.05	Move Package	59	0.05
Add Attribute Modifier	1141	1.05	Rename Package	58	0.05
Add Thrown Exception Type	1073	0.99	Merge Variable	52	0.05
Change Attribute Access Modifier	1011	0.93	Add Variable Annotation	51	0.05
Modify Class Annotation	944	0.87	Replace Anonymous With Class	51	0.05
Move Attribute	910	0.84	Inline Attribute	50	0.05
Pull Up Method	858	0.79	Replace Pipeline With Loop	47	0.04
Extract Attribute	844	0.78	Extract Interface	32	0.03
Remove Variable Modifier	781	0.72	Merge Attribute	31	0.03
Remove Parameter Modifier	711	0.66	Merge Method	28	0.03
Modify Attribute Annotation	700	0.65	Push Down Attribute	28	0.03
Merge Conditional	634	0.58	Merge Catch	27	0.02
Rename Class	631	0.58	Move And Rename Attribute	19	0.02
Remove Parameter Annotation	554	0.51	Split Parameter	18	0.02
Remove Method Modifier	509	0.47	Change Type Declaration Kind	15	0.01
Encapsulate Attribute	503	0.46	Extract Subclass	10	0.01
Inline Method	497	0.46	Merge Class	10	0.01
Change Thrown Exception Type	445	0.41	Split Class	9	0.01
Add Attribute Annotation	423	0.39	Replace Attribute	6	0.01
Pull Up Attribute	416	0.38	Split Attribute	4	0.00
Add Method Modifier	412	0.38	Split Package	4	0.00

Table 4: Distribution of Refactorings by Family Type (RQ_1)

Refactoring Family	**#**	**%**	**Refactoring Family**	**#**	**%**
Add	23597	21.74	Pull	1274	1.17
Change	22221	20.48	Merge	900	0.83
Remove	15133	13.94	Encapsulate	503	0.46
Rename	14762	13.60	Assert	287	0.26
Extract	12064	11.12	Invert	265	0.24
Move	7074	6.52	Split	248	0.23
Modify	4402	4.06	Localize	192	0.18
Replace	2051	1.89	Push	155	0.14
Inline	1859	1.71	Try	139	0.13
Parameterize	1287	1.19	Reorder	110	0.10

Moreover, averaging across RF, although RE persists for less than 10 subsequent refactorings, regardless of RF (Figure 8), there is considerable variance in both the distribution and outliers. Refactorings in the Add, Change, and Move families exhibit the widest variation. Notably, Add-based refactorings can persist through more than 190 subsequent refactorings, while Change- and Move-related refactorings can last as long as 170 subsequent refactorings. Considering the observed wide duration of the RE, we were keen to investigate how much of the original refactoring lasted during the RE.

Finally, since all our designed hypotheses rely on the concept of time in terms of RE duration, we computed its distribution. We organized the results into quartiles to facilitate further interpretation:

– Q_1 (1–4 consecutive refactorings)
– Q_2 (4–9 consecutive refactorings)
– Q_3 (9–19 consecutive refactorings)
– Q_4 (19–191 consecutive refactorings)

Before testing our hypothesis, according to our study methodology (Section 3.4), we must test the data distribution. Therefore, to assess whether RE is normally distributed, we tested $H_{0\mathcal{N}1}$, and we could reject it (A^2 12719,677072, p-value $< 0,0001$).

To assess whether the duration of RE is significantly different among RF, we tested H_{01}, and we could reject it in 13 out of 20 refactoring types (Table 5). Therefore, we can affirm that **for most of the refactoring families, the difference in the RE duration is statistically significant**. Finally, we could not compute WT in 6 out of 20 cases ("-") since there was only one subtype in such RFs, i.e., 0 degree of freedom. Combining the findings of Figure 8 and Table 5, the refactoring families for which the RE exhibited many extreme cases are also statistically significant. . Consequently, to assess whether the differences in the RE duration between RFs are statistically significant, we tested H_{02}, and we could reject it in 116 over 190 pairs (Table F.2). Therefore, we can also affirm that **most of the differences in the RE duration between refactoring families are statistically significant**. More specifically, RF, such as Invert and Split, have the highest mean difference in RE duration compared to other families (e.g.,

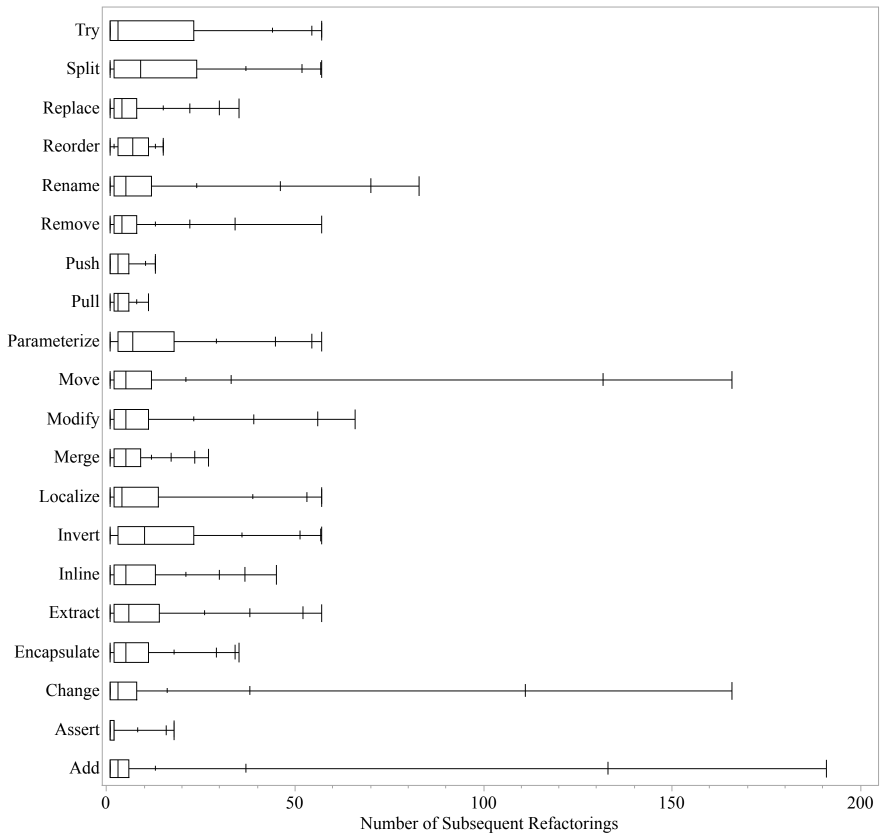

Fig. 8: Distribution of RE duration by refactoring family type (RQ$_1$)

Table 5: Wilcoxon Test Results for RE Duration Across Refactoring Families (RQ$_1$ - H_{01})

Refactoring Family	p-value	Refactoring Family	p-value
Add	<.0001	Move	<.0001
Assert	·	Parameterize	<.0001
Change	<.0001	Pull	0.0023
Encapsulate	·	Push	<.0001
Extract	<.0001	Remove	<.0001
Inline	0.1225	Rename	<.0001
Invert	·	Reorder	·
Localize	·	Replace	<.0001
Merge	<.0001	Split	<.0001
Modify	<.0001	Try	·

Table 6: Wilcoxon Test on the Differences of LCS Observation Over Time by RF (RQ_1 - H_{03})

Refactoring Family	ChiSquare	DF	p-value	Refactoring Family	ChiSquare	DF	p-value
Add	5674.6894	190	<0.0001	Modify	1279.2289	65	<0.0001
Assert	109.2006	17	<0.0001	Move	2302.8577	165	<0.0001
Change	5733.3615	165	<0.0001	Parameterize	479.9441	56	<0.0001
Encapsulate	81.5739	34	<0.0001	Pull	158.5907	10	<0.0001
Extract	4418.0128	56	<0.0001	Push	81.6112	12	<0.0001
Inline	564.5407	44	<0.0001	Remove	3179.5227	56	<0.0001
Invert	67.3806	56	0.1418	Rename	3808.1002	82	<0.0001
Localize	134.8105	56	<0.0001	Reorder	33.7462	14	0.0022
Merge	215.9336	26	<0.0001	Replace	307.0007	34	<0.0001
Modify	1279.2289	65	<0.0001	Split	57.4565	56	0.4210
Move	2302.8577	165	<0.0001	Try	103.3575	56	0.0001

Assert and Add). Rename and Extract families also present high differences, suggesting that naming convention change and method restructuring are important in RE duration. Conversely, refactorings such as Replace and Modify present lower mean differences, suggesting a more localized effect. Dunn's test results highlight that structural changes have longer-lived RE, while code-level changes produce shorter-lived effects.

Finally, recalling our definition of RE, we were keen to assess the remaining share of code that remained intact from the original refactoring using Bayesian probabilities; however **its result was unsatisfactory**; therefore, we report the result of the alternative method (LCS) (see Section 3.4). In cases where RE lasts between one and six subsequent refactorings (Figure 9 (a-b)), the average LCS ranges from 80% to 100%, hinting that refactorings appear to introduce code changes robust enough to persist reliably through four and up to six subsequent refactorings despite numerous outliers exhibiting significantly lower LCS. Conversely, when examining the third quartile (Figure 9(c)), the average LCS concerning the original refactoring operation decreases notably, ranging between 40% and 60%. This indicates a trend toward lower similarity, although considerable variation remains. More specifically, refactoring types under the Localize, Assert, and Reorder families have the lowest LCS, ranging between 20 and 40%. We can notice a general shift towards lower percentages in the last quartile (Figure 9(d)), with most of the refactoring families having their mean similarity between 20 and 40% and a more compact distribution.

Similarly, for RE duration, we must test LCS distribution via $H_{0\mathcal{N}2}$, and we could reject it (A^2 12719,677072, p-value $< 0,0001$). Therefore, to assess whether LCS significantly changes over RE duration, we test H_{03} and rejected it in 18 out of 20 RF. **Therefore, we can affirm that the differences in LCS observations over RE duration are statistically significant**.

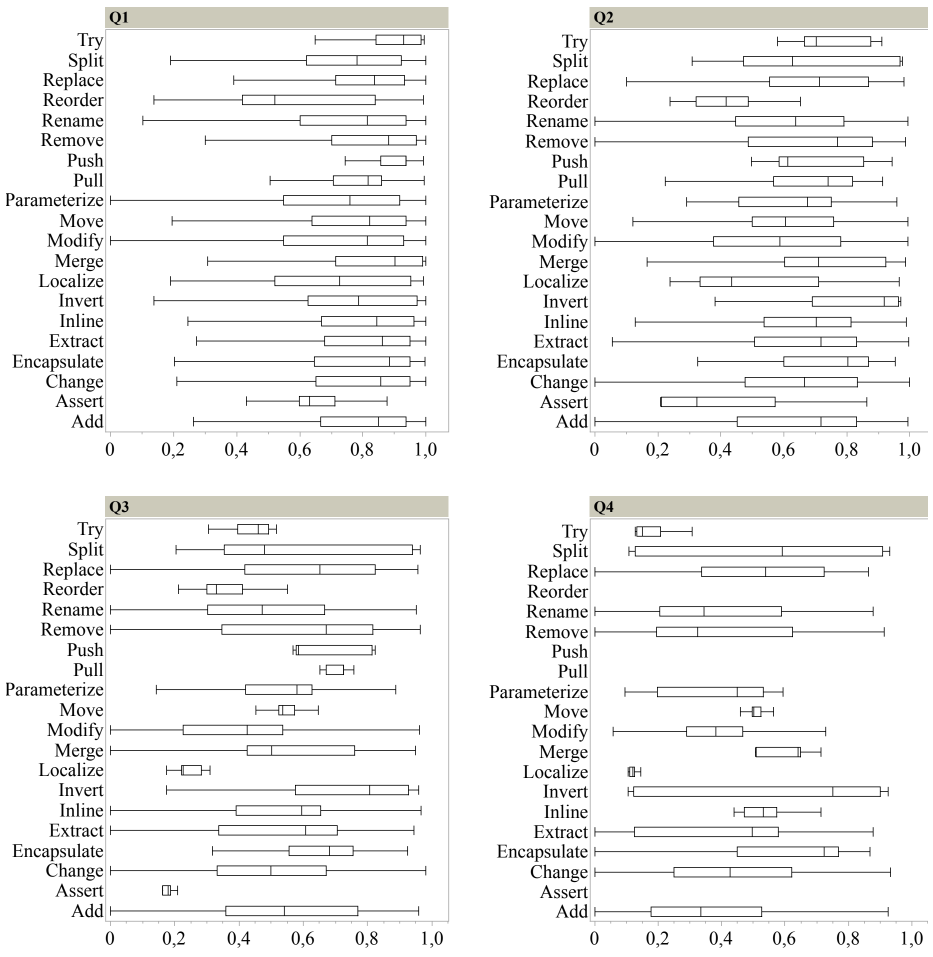

Fig. 9: LCS Distribution by RE duration quartiles (RQ_1)

1. **Key Take Away**: **RE of a refactoring activity typically persists for fewer than 10 subsequent refactorings**. However, the distribution is broad, and the difference between family types is statistically significant, and extreme cases exceed 190 refactorings. Structural refactorings (e.g., Split, Invert) have longer-lived RE, while localized changes (e.g., Replace, Modify) fade faster. Initially, 80–100% of refactored code remains intact, but similarity drops to 20–40% over time rapidly depending on the overall RE and are statistically significant.

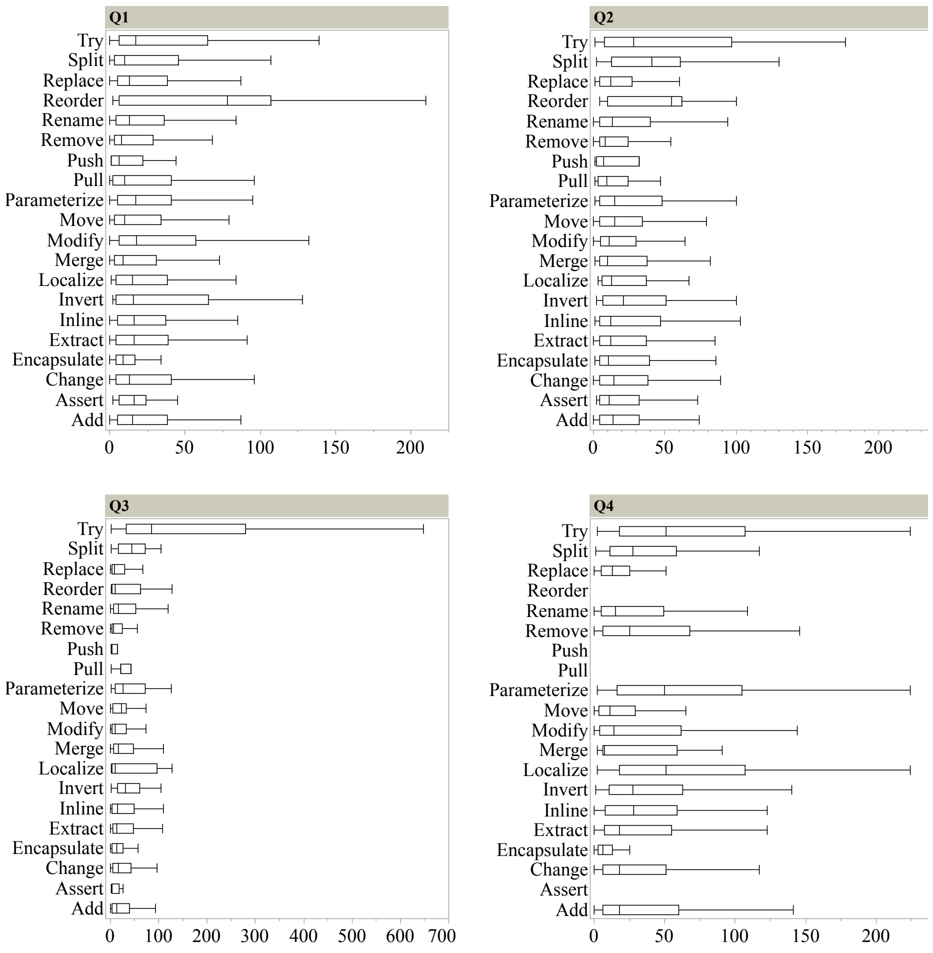

Fig. 10: Change Proneness Distribution by RE duration quartiles (RQ_2)

4.2 The long-term effect of refactoring on change and defect proneness (RQ_2)

In this section, we present our findings on the influence of the RE over the change ($\mathcal{C}$) and defect proneness ($\mathcal{D}$). We observe that $\mathcal{C}$ varies the most in Q_1, Q_2, and Q_4 while exhibiting a stable though very narrow distribution in Q_3 (Figure 10), hinting that refactorings that have a RE duration between 9 and 19 are the ones that are less prone to change again. Besides, Assert, Pull, Push, and Reorder show a decreasing $\mathcal{C}$, resulting in no refactoring after Q_3. Therefore, we note that the $\mathcal{C}$ shows a decreasing $\mathcal{C}$ with more subsequent refactoring for each refactoring family. Conversely, Q_4 represents where the outliers lie and it is unsurprising to observe the broader ranges of $\mathcal{C}$ RFs.

To assess whether the differences observed in $\mathcal{C}$ are significant, we first must test $\mathcal{C}$ distribution via $H_{0\mathcal{N}3}$, and we could reject it (A^2 32312,173457, p-value $< 0,0001$). We then tested H_{04} and we could reject it in 16 out of 20 cases

Table 7: Wilcoxon Test Results on the Differences of $\mathcal{C}$ Observation Over Time by RF (RQ_2 - H_{04})

Refactoring Family	ChiSquare	DF	p-value	Refactoring Family	ChiSquare	DF	p-value
Add	1097.5763	190	<0.0001	Move	937.9921	165	<0.0001
Assert	165.0874	17	<0.0001	Parameterize	356.0281	56	<0.0001
Change	623.1438	165	<0.0001	Pull	166.6922	10	<0.0001
Encapsulate	85.5960	34	<0.0001	Push	61.2158	12	<0.0001
Extract	715.4987	56	<0.0001	Remove	616.0260	56	<0.0001
Inline	244.2007	44	<0.0001	Rename	548.8002	82	<0.0001
Invert	60.6522	56	0.3119	Reorder	39.2957	14	0.0003
Localize	71.9896	56	0.0737	Replace	69.8012	34	0.0003
Merge	64.5427	26	<0.0001	Split	75.0444	56	0.0455
Modify	406.9581	65	<0.0001	Try	69.8739	56	0.1006

(Table 7), thus **the differences in $\mathcal{C}$ observation across RF by RE duration is statistically significant**.

Interestingly, regarding $\mathcal{D}$, "Add," "Change," "Extract," "Remove," and "Rename" RFs are consistently the most defect-prone across quartiles (Figure 11). Furthermore, from Q_1 to Q_3, all refactoring families follow a similar trend, predominantly defect-prone rather than defect-free. Quartile Q_4 presents the most variable distribution, characterized by a notable increase in defect-prone refactorings. It suggests that the longer a refactoring endures, i.e., the longer RE persists, the greater the likelihood it will become defect-prone.

Similarly, to assess whether the differences observed in $\mathcal{D}$ are significant, we first must test $\mathcal{C}$ distribution via $H_{0\mathcal{N}4}$, and we could reject it (A^2 2102,9563651, p-value $< 0,0001$). Then, we tested H_{05}, and we could reject it in 15 out of 20 cases (Table 8); thus, **the differences in $\mathcal{D}$ observation across RF by RE duration is statistically significant**. We note that both for $\mathcal{C}$ and $\mathcal{D}$, the non-statistically significant families are Invert, Localize, Push, Split, and Try, which are also the least represented refactoring families according to Table 4.

Finally, we were keen to assess whether $\mathcal{C}$ and $\mathcal{D}$ observations are correlated with RE duration. Therefore, to assess the correlation between $\mathcal{C}$ and RE duration, we tested $H_{0\rho 1}$, and we could reject it in 17 of 20 cases (Table 9), thus **$\mathcal{C}$ observations across RF are statistically correlated with RE duration**. More specifically, the highest positive correlation is in the "Try" refactoring family ($\rho =$ 0.3338), with "Assert" being very close to it ($\rho = 0.3188$), both being statistically significant ($p < 0.0001$). In contrast, the highest negative correlation is that of "Reorder" ($\rho =$ -0.1972, p = 0.0389). All refactoring families but "Encapsulate" and "Invert" have statistically significant correlations (ρ <0.01), indicating no significant relation between RE and $\mathcal{C}$ for these families.

Similarly, to assess the correlation between $\mathcal{D}$ and RE duration, we tested $H_{0\rho 2}$, and we could reject it in 13 out of 20 cases (Table 10), thus **$\mathcal{D}$ observations across RF are statistically correlated with RE duration**. The "Parameterize" RF exhibited the highest positive correlation ($\rho = 0.2656$), with a significant correlation between RE and $\mathcal{D}$. In contrast, the "Encapsulate" family features the lowest correlation ($\rho =$ -0.0391), albeit not statistically significant (p = 0.3821). Except

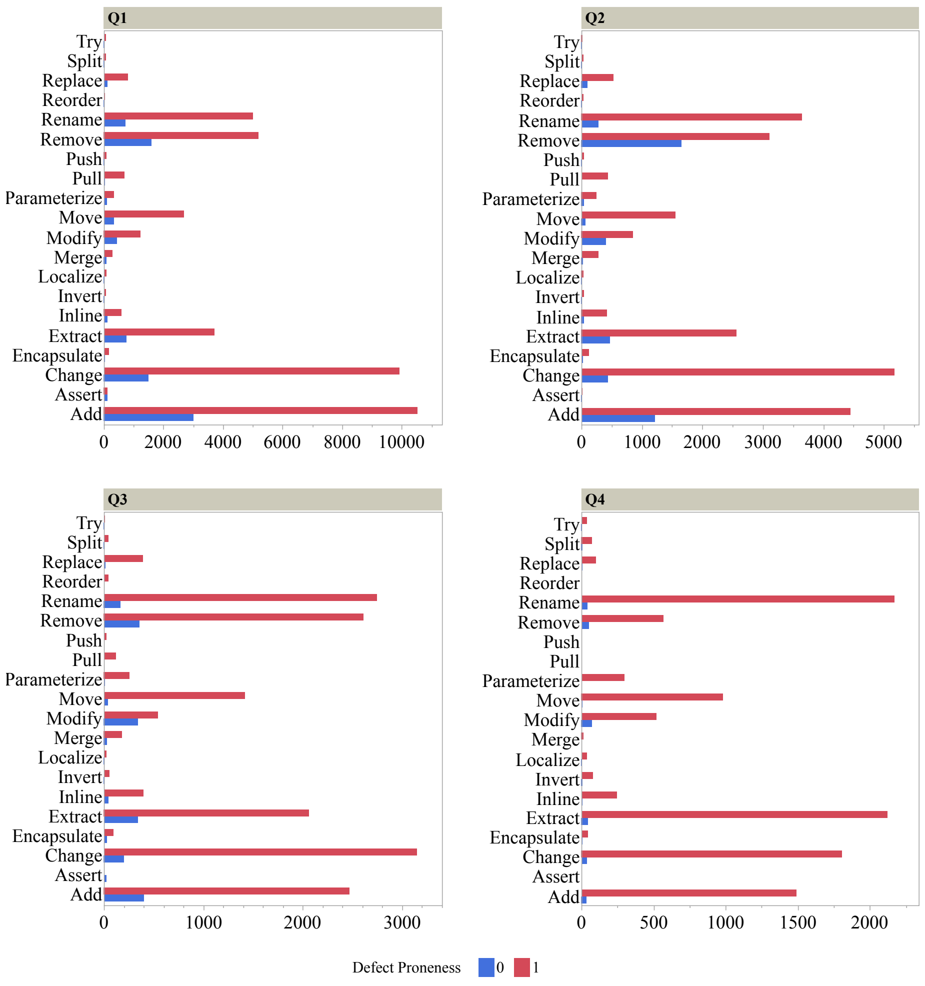

Fig. 11: Defect Proneness Distribution by RE duration quartiles (RQ_2)

for six, all the correlations are statistically significant, indicating no notable relationship between RE and $\mathcal{D}$ in the case of "Encapsulate, "Localize," "Modify," "Push," "Reorder," and "Try."

2. **Key Take Away**: RE significantly impacts change- and defect-proneness over time. Medium-term refactorings, i.e., Q_2 (9–19 subsequent changes), are the most stable, showing low change-proneness. Longer-lasting refactorings become increasingly defect-prone. Families such as "Try," "Assert," and "Parameterize" are particularly sensitive to this effect, whereas "Encapsulate" has minimal long-term impact.

Table 8: Wilcoxon Test Results on the Differences of $\mathcal{D}$ Observation Over Time by RF (RQ$_2$ - H_{05})

Refactoring Family	**ChiSquare**	**DF**	**p-value**	**Refactoring Family**	**ChiSquare**	**DF**	**p-value**
Add	676.2689	190	<0.0001	Move	241.1988	165	0.0001
Assert	155.7211	17	<0.0001	Parameterize	176.7791	56	<0.0001
Change	453.4296	165	<0.0001	Pull	24.9881	10	0.0054
Encapsulate	126.7689	34	<0.0001	Push	13.8277	12	0.3118
Extract	523.5042	56	<0.0001	Remove	885.2277	56	<0.0001
Inline	81.3291	44	0.0005	Rename	468.3467	82	<0.0001
Invert	64.8133	56	0.1963	Reorder	49.2947	14	<0.0001
Localize	39.2511	56	0.9564	Replace	91.1844	34	<0.0001
Merge	79.6698	26	<0.0001	Split	60.3490	56	0.3215
Modify	299.1744	65	<0.0001	Try	30.9436	56	0.9974

Table 9: Spearman Correlation between Change-Proneness and Ripple Effect across Refactoring Families (RQ$_2$ - $H_{0\rho 1}$)

Refactoring Family	**Spearman ρ**	**p-value**	**Refactoring Family**	**Spearman ρ**	**p-value**
Add	0.0229	0.0004	Move	0.0647	<0.0001
Assert	0.3188	<0.0001	Parameterize	0.2391	<0.0001
Change	0.0366	<0.0001	Pull	0.0913	0.0011
Encapsulate	0.0109	0.8079	Push	0.2351	0.0032
Extract	0.0532	<0.0001	Remove	0.0332	<0.0001
Inline	0.0898	0.0001	Rename	0.0625	<0.0001
Invert	0.1057	0.0859	Reorder	-0.1972	0.0389
Localize	0.2773	<0.0001	Replace	-0.0687	0.0019
Merge	0.1521	<0.0001	Split	0.1946	0.0021
Modify	-0.09	<0.0001	Try	0.3338	<0.0001

4.3 The long-term effect of refactoring benefit/effort ratio (RQ$_3$)

In this section, we present our findings on the influence of the RE over the change efficiency ($\mathcal{CE}$). We observe that $\mathcal{CE}$ varies the most in Q$_2$, and Q$_4$ while exhibiting a stable though very narrow distribution in Q$_1$ and Q$_3$ (Figure 12). Regarding the latter, Q$_1$ and Q$_3$, we refer to RE between 1-4 and 9-19 subsequent refactoring, hinting that such RE lengths are the ones that produce more stable refactors. According to our definition of $\mathcal{CE}$, a low ratio value suggests that such refactoring leads to a subsequent higher number of changes with a lower effort (ELOC). Such results are also confirmed by RQ$_2$, in which refactoring falls below the Q$_3$ of RE, which is less prone to further changes. More specifically, for Q_1 Assert, Invert, Merge, and Split have the lowest ratio, while Reorder, Move, and Modify are the least stable, hence hinting that such refactoring activities are more prone to lead to subsequent changes which require more effort to accomplish. We must highlight

Table 10: Spearman Correlation between Defect-Proneness and Ripple Effect across Refactoring Families (RQ$_2$ - $H_{\rho 01}$)

Refactoring Family	**Spearman ρ**	**p-value**	**Refactoring Family**	**Spearman ρ**	**p-value**
Add	0.1206	<0.0001	Move	0.1691	<0.0001
Assert	0.2541	<0.0001	Parameterize	0.2656	<0.0001
Change	0.1209	<0.0001	Pull	0.0591	0.0348
Encapsulate	-0.0391	0.3821	Push	-0.0241	0.7656
Extract	0.1316	<0.0001	Remove	0.0700	<0.0001
Inline	0.1275	<0.0001	Rename	0.1386	<0.0001
Invert	0.2427	<0.0001	Reorder	0.1350	0.1596
Localize	0.0392	0.5893	Replace	0.0837	0.0001
Merge	0.1210	0.0003	Split	0.1907	0.0026
Modify	-0.0092	0.5437	Try	0.1241	0.1456

that Reorder shows the highest value, with outliers hitting ratios as high as 1.4. Similarly, in Q$_3$, Assert, Move, Pull, and Push have the lowest ratio, while Change, Encapsulate, Localize, Rename, and Try to have the most unstable.

Refactorings falling in the RE's Q$_2$ have an average low ratio with no notable difference except for a spike in the Try RF that is also present in Q$_3$.

Regarding interesting patterns, we note the Try RF that induces stable refactorings Q$_1$ and rapidly increases between Q$_2$ and Q$_3$ while decreasing again in Q$_4$. Therefore, we can speculate that such refactoring activity leads to an increased effort in terms of lines of code for fewer changes. Conversely, the Reorder RF induces the exact opposite phenomenon. Hence, we speculate that Reorder refactorings lead to more complex changes in the source code in the short time but are the ones that benefit more in the long run, disappearing in Q$_4$. Finally, the Change RF presents a constant, low ratio behavior throughout the quartiles.

To assess whether the differences observed in $\mathcal{D}$ are significant, we first must test $\mathcal{C}$ distribution via $H_{0\mathcal{N}5}$, and we could reject it (A^2 26887,544, with p-value $< 0,0001$). Then, we tested H_{06}, and we could reject it in 16 out of 20 cases (Table 11); thus, **the differences in $\mathcal{CE}$ observation across RF by RE duration is statistically significant**. Most refactoring families show statistically significant differences in $\mathcal{CE}$ across time, with p-values consistently below 0.0001. This includes prominent families such as Add, Change, Move, and Rename, which also report the highest ChiSquare values—exceeding 1000 in several cases—suggesting strong effects. The refactorings Extract, Remove, and Modify also show substantial ChiSquare values, reinforcing the temporal influence on these operations' $\mathcal{CE}$ observations.

On the other hand, a few refactoring families—namely Invert, Localize, Split, and Try—do not exhibit statistically significant changes over time, as indicated by p-values well above the 0.05 threshold. Among them, Try has the highest p-value (0.3940), followed by Invert (0.3282), indicating the weakest evidence against the null hypothesis of no change. Notably, Merge is the only borderline case, with a p-value of 0.0035, which is still statistically significant but with a relatively low ChiSquare value, suggesting a more moderate effect.

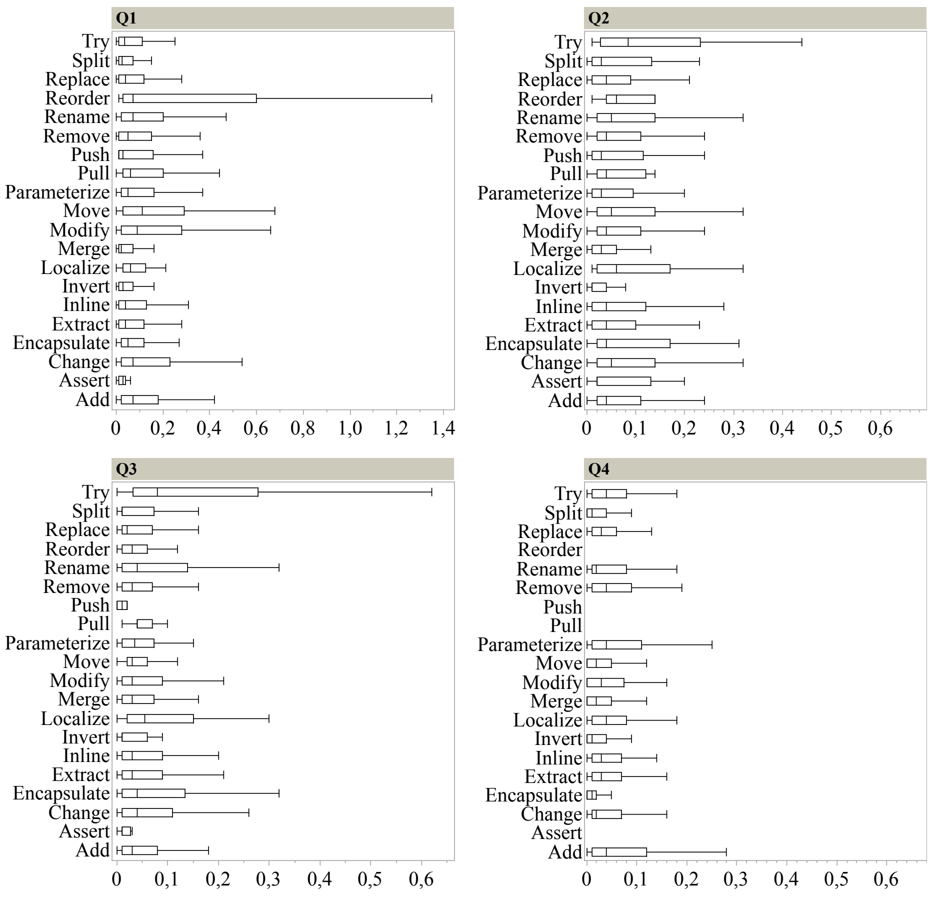

Fig. 12: Change Efficiency Distribution by RE duration quartiles (RQ$_3$)

Table 11: Wilcoxon Test Results on the Differences of $\mathcal{CE}$ Observation Over Time by RF (RQ$_3$ - H_{06})

Refactoring Family	ChiSquare	DF	p-value	Refactoring Family	ChiSquare	DF	p-value
Add	1666.0979	190	$< .0001$	Move	1843.6245	165	$< .0001$
Assert	121.4613	17	$< .0001$	Parameterize	291.5744	56	$< .0001$
Change	1713.8427	165	$< .0001$	Pull	182.6913	10	$< .0001$
Encapsulate	98.3447	34	$< .0001$	Push	72.3857	12	$< .0001$
Extract	620.0719	56	$< .0001$	Remove	735.3810	56	$< .0001$
Inline	214.0620	44	$< .0001$	Rename	1067.8906	82	$< .0001$
Invert	60.1418	56	0.3282	Reorder	42.6542	14	$< .0001$
Localize	62.5451	56	0.2552	Replace	101.0995	34	$< .0001$
Merge	49.6468	26	0.0035	Split	64.9770	56	0.1924
Modify	655.4453	65	$< .0001$	Try	58.2085	56	0.3940

3. **Key Take Away**: Refactorings are most stable, i.e., low $\mathcal{CE}$, for short and medium-long refactoring episodes (Q_1 and Q_3), suggesting they yield better long-term benefits. Refactorings like Assert and Merge are more stable, while Reorder and Modify often lead to higher effort. Try shows fluctuating behavior—stable early, peaking mid-way, then declining. Statistical analysis confirms significant variation in $\mathcal{CE}$ over time for most refactorings, except a few like Try and Invert.

5 Discussion

This section discusses the findings regarding the persistence of the refactoring effect (RE) and its long-term implications on change- and defect-proneness.

Our results for RQ_1 show significant variation in the persistence of RE across different refactoring families. The refactoring operations within the **Add** and **Change** families, accounting for approximately 40% of all refactoring operations, suggest a tendency towards operations focusing on code enhancement rather than extensive restructuring. The large variability in the RE duration, particularly in the Add, Change, and Move families, underscores that specific refactorings can persist significantly, sometimes beyond 190 subsequent refactorings, indicating their long-lasting impact on codebases.

1. **RE Duration Variability**: Practitioners should focus on refactoring in the "Add," "Change," and "Move" families early in the development lifecycle. Such families are more likely to produce long-lasting changes that may affect future maintainability. Similarly, researchers should focus on developing a tool to support and guide developers in assessing the long-term implications of refactorings.

Furthermore, the statistical tests validate the significance of differences between refactoring families regarding RE duration. Specifically, structural refactoring types, i.e., from the "Split" and "Invert" families, have effects that last longer. Such a result is consistent with the intuitive expectation that structural refactoring, involving substantial code reorganization or architectural transformation by construction, has a lasting impact on subsequent software development.

LCS confirms that refactoring-introduced code is highly similar at the start (80–100%) but continues to decrease, particularly after the 6th subsequent refactoring. Surprisingly, with the longer persistence of refactoring effects, the code similarity sharply drops to around 20–40%, strengthening the diminishing retention of originally introduced refactoring code over extended development periods.

2. **LCS Duration**: The natural decrease in code similarity between the original refactor and the last in the chain suggests that practitioners should periodically revisit long-lived refactorings to validate whether their intended value still holds. Similarly, researchers should focus on tracking and visualizing

the historical footprint of refactorings across code versions to evaluate their sustained effectiveness.

For the long-term effect on software maintainability (RQ$_2$), CP analysis shows that refactorings with medium-term persistence (lasting between 9 and 19 subsequent refactorings) are the least likely to change again. This stability suggests **the presence of a time window during which refactoring is most beneficial, after which its beneficial effects wear off or even become harmful**.

☞ 3. **Medium-term benefit**: Practitioners should monitor refactoring that persists beyond approximately 20 changes, as their initial benefits may turn into liabilities. RE computation can be integrated into CI/CD workflows to flag ageing refactorings that may require reassessment. Such a result will pave the way to design predictive models that proactively identify long-lived resources at risk of degradation and maintainability.

DP analysis also reveals **the danger of lengthy RE durations**. Refactorings that last longer than 19 subsequent operations are more defect-prone, and this is especially evident in refactoring families such as “Try,” “Assert,” and “Parameterize.” These families exhibit high positive correlations between RE duration and DP, suggesting that specific refactorings, especially those involved with complex code manipulation or error handling, are hazardous over long periods.

Interestingly, refactoring families like “Encapsulate,” “Localize,” and “Modify” have small or statistically insignificant long-term impacts, pointing to their promise of being safe, incremental improvements with little chance of introducing defects or necessitating further code changes.

☞ 4. **Defect-prone Try/Assert/Parameterize**: Refactorings such as Try and parametric, exhibiting high DP over time, should be treated carefully. On the one hand, practitioners are encouraged to carefully review their use, especially when the source code is refactored across many ranges. On the other hand, researchers should explore improving refactoring tools with a risk-aware recommendation and alerting mechanism, defining a taxonomy of refactoring criticalities.

Therefore, we can identify a specific lesson learned for practitioners: **structural refactorings are more powerful but entail increased long-term risk and maintenance cost**. Conversely, localized refactorings yield more predictable short-term benefits at reduced risk. Our results can help developers and project managers decide when and what type of refactoring activities must be carried out to enhance software quality and maintainability best.

☞ 5. **Challenges of Structural Refactoring**: Due to their high and lasting impact, practitioners should carefully plan structural refactoring, ideally early in the release cycle and coupled with a thorough code review. Researchers should focus on determining techniques that may support timing, planning, and risk assessment of structural refactoring.

$\mathcal{CE}$ presents a solid point of view to quantify the long-term benefit of a refactoring action regardless of its short-term structural impact. Early on, our findings show that the most effective and robust refactorings, those with more significant change for less effort, are concentrated in Q_1 and Q_3, which cover refactoring spans of 1–4 and 9–19 consecutive changes, respectively. This is also a reflection of our earlier result in RQ_2, where Q_3 refactorings also exhibited the lowest change-proneness ($\mathcal{C}$), pointing towards a sweet spot during the lifecycle of refactoring when the created change is durable and worthwhile but not overwhelming. Similarly, RQ_1 had already suggested that RE typically persists for fewer than ten subsequent refactorings, which aligns with the time when $\mathcal{CE}$ is the most stable.

But when we look at the outliers, most notably Q_4, in which refactorings survive longer than 19 changes, recalling that RQ_1 showed that particular REs lasted longer than 190 operations, the equivalent $\mathcal{CE}$ in such quartile grows increasingly unstable. This instability mirrors RQ_2 outcomes, where more long-lived refactorings lean more towards being more defect-prone. The increased volatility in $\mathcal{CE}$ at these extended lifespans likely results from a trade-off. As code evolves, even well-meaning, motivated refactorings will incur technical debt or make cascading changes disproportionately expensive to repair.

Focusing on each family of refactorings reveals yet finer patterns. "Assert," "Merge," and "Split" refactorings demonstrate high stability in early RE times (Q_1), indicating low-cost, high-benefit interventions. Mainly, Assert is unique in all the RQs, it not only exhibits high code similarity persistence in RQ_1 and low change-proneness in RQ_2, but also returns comparable benefit-to-effort ratios in RQ_3. Reorder, though an initial instability and high $\mathcal{CE}$ expense, also "pays off" in the end, disappearing nearly entirely in Q_4 for an initial cost with long-term architectural stability.

Another fascinating example is the "Try" family of refactorings. Its quartile performance is almost sinusoidal, steady in Q_1, rising to peak complexity in Q_2 and Q_3, then declining in Q_4. This trend reflects a productive first half, followed by a muddled middle part before returning to equilibrium. These trends emphasize that not all forms of refactoring follow a linear payoff curve; their lifecycle impact is moderated by a mix of their structural nature and contextual use within the codebase. Finally, structural refactorings with longer RE durations have a higher chance of incurring higher maintenance costs for defects and the effort necessary. Yet some well-scope and suitably applied refactorings i.e., particularly ones of medium RE duration, are resilient and cost-effective over the long run. This cross-point of persistence, stability, and efficiency constitutes an interesting lesson learned for practitioners: **most valuable refactorings are not the most persistent but those whose durations fall within some middle point between low change-proneness and high benefit-to-effort ratio.**

☞ 6. $\mathcal{CE}$ **and Q1/Q3 "sweet spot"**: Medium-duration refactorings (Q_3) represent a sweet spot regarding long-term cost-effectiveness and stability. Practitioners could prioritize such refactorings during maintenance activities, guided by effort/benefit heuristics like our $\mathcal{CE}$. Conversely, researchers should explore predictive models and smart tooling to identify and recommend refacing in the optimal time window proactively.

Our findings offer promising avenues for future research and practical software engineering practice. One of the interesting avenues of research goes beyond typical refactoring recommendations to the development of predictive models that can forecast how long the impact of a given refactoring will endure. Such models would enable practitioners to forecast under what conditions a refactoring becomes useful or even hazardous over its lifespan. In addition, current refactoring tools, i.e., embedded in IDEs or CI/CD pipelines, would significantly benefit from adding risk-conscious recommendations. With RE-informed warnings, tools can alert developers ahead of a planned refactoring when it might affect maintainability or introduce defect-proneness. Another promising avenue is to improve how researchers compare refactoring tools.

Generally, current benchmarks care about nothing but temporary success, like code smell removal quality or temporary syntactic correctness. However, our results suggest that longitudinal benchmarking standards are necessary to measure the long-term effects of refactorings on software well-being and sustainability. From a pragmatic perspective, our results encourage software development teams to investigate the incorporation of RE-aware recommendations as part of their local coding principles.

For instance, refactoring on highly volatile components should preferably be directed towards shorter-lived, lower-risk ones. At the same time, more stable, long-lived modules may better cope with more significant structural interventions, especially if these are strategically scheduled early in the release cycle.

💡 4. **Key Take Away**: The most valuable refactorings are not necessarily the most persistent, but those that achieve a balance, lasting long enough to deliver meaningful improvements while avoiding long-term risk. Practitioners should favor medium-term refactorings that offer this trade-off. Researchers are encouraged to develop tool-supported strategies that time and recommend refactorings based on their predicted lifecycle.

Finally, RE-aware knowledge can facilitate improved communication between developers and maintainers so that short-term coding decisions correlate with long-term software sustainability. By carrying out refactoring operations with a clear understanding of their long-term impacts, developers can better describe their motives and expected consequences to maintainers, who become better qualified to sustain, analyze, and add to the refactored code. By including an explicit consideration of persistence refactoring, risk, and maintainability effects in development meetings, teams make sure that their immediate technical decisions align with overall design strategies and long-term sustainability goals. This improved correspondence improves coordination among team members and provides a helpful anchor for the empirical study of how co-working developers come to a consensus about refactoring decisions. Thus, developing tools and frameworks to support this form of RE-informed communication is a significant line of future work and practical development in software engineering.

6 Guidelines for Developers

Our study provides not only new empirical evidence on the ripple effects of refactoring but also clear recommendations that developers can adopt in practice. Below, we describe the main lessons learned and highlight them in concise guidelines.

Refactorings are not equally sustainable over time. Our results demonstrate that medium-lived refactorings, typically lasting between 9 and 19 subsequent changes, provide the most stable improvements. They combine low change-proneness with favorable benefit-to-effort ratios. By contrast, very long-lived refactorings, although initially impactful, often turn into liabilities and raise maintenance costs.

1. **Favor medium-lived refactorings**: Medium-lived refactorings (9–19 changes) maximize long-term benefits while avoiding the risks of overly persistent refactorings.

Structural refactorings, such as *Split* or *Invert*, can have profound effects that persist for more than 190 subsequent changes. However, their long-term influence comes with a higher risk of defect-proneness and elevated maintenance effort. This means that such interventions should not be avoided altogether, but carefully planned. In particular, they are more beneficial when executed early in the release cycle and supported by rigorous reviews.

2. **Plan structural refactorings carefully**: Structural refactorings should be scheduled strategically and paired with reviews, given their higher defect-proneness.

At the other end of the spectrum, smaller refactoring families like *Encapsulate*, *Localize*, and *Modify* proved to be reliable, showing little to no harmful long-term impact. These localized interventions offer developers a safe way to achieve incremental improvements, particularly in volatile code regions where major structural changes would be risky.

3. **Use localized refactorings for safe improvements**: Localized refactorings (*Encapsulate*, *Localize*, *Modify*) enable low-risk, incremental code quality gains.

Another key lesson is that refactorings persisting beyond twenty changes frequently become problematic. Families such as *Try*, *Assert*, and *Parameterize* showed strong correlations with defect-proneness over long lifespans. Teams should therefore monitor such refactorings continuously, ideally using CI/CD alerts, to reassess them before they accumulate technical debt.

📘 4. **Monitor long-lived refactorings**: Refactorings lasting 20+ changes require proactive monitoring to prevent defect accumulation and technical debt.

Finally, our benefit/effort analysis emphasizes that refactoring should not be carried out in isolation from the development roadmap. Refactorings aligned with anticipated feature additions and long-term system evolution consistently provided higher benefits. Developers gain more value when they refactor not simply to clean up code, but to proactively enable smoother maintenance and easier implementation of future changes.

📘 5. **Align refactoring with evolution plans**: Refactor with future system evolution in mind to reduce rework and ensure long-term maintainability.

It is equally important to emphasize not only when refactoring is beneficial, but also when it should be avoided. Our results reveal that extremely persistent refactorings, especially those beyond twenty subsequent changes, often become liabilities rather than assets. In such cases, the increased defect-proneness and maintenance burden outweigh the intended benefits. Thus, developers should refrain from performing large-scale or highly invasive refactorings unless they are strictly necessary to resolve critical architectural debt.

📘 6. **Guidelines**: Avoid unnecessary long-lived refactorings. Very persistent refactorings (20+ changes) frequently turn into liabilities and should be avoided unless addressing critical architectural issues.

Beyond the individual lessons, our findings also carry practical implications for development workflows. Teams can operationalize these guidelines by: (i) monitoring refactoring persistence and defect-proneness using automated mining tools, (ii) integrating refactoring alerts into CI/CD pipelines to flag risky, long-lived interventions, and (iii) aligning refactoring decisions with backlog planning to ensure that code improvements directly support upcoming features. These practices help developers apply our guidelines in a systematic way and transform empirical insights into everyday engineering routines.

7 Threats to Validity

In this section, we discuss the threats to the validity of our study. We categorized the threats into Construct, Internal, External, and Conclusion, following the guidelines defined by Wohlin et al. (Wohlin et al., 2012).

Construct Validity. Construct validity concerns how our measurements reflect what we claim to measure (Wohlin et al., 2012). Our specific design choices, including our measurement process and data filtering, may impact our results. To address this threat, we based our choice on past studies (Palomba et al., 2018;

Sellitto et al., 2022) and used well-established guidelines in designing our methodology (Basili et al., 1994; Runeson and Höst, 2009). We further acknowledge that the use of tools like RefactoringMiner for detecting refactorings and regular expressions for issue-fixing commit extraction introduces a level of approximation. While these tools are widely validated and accepted in the literature, they may introduce false positives or negatives. Nevertheless, we have followed the previous research methodology to mitigate such threats, and consequently leading this issue to be a minor threat to validity. Similarly, we acknowledge the adoption of metrics used as a proxy to measure developer effort, that might not fully capture other dimensions related to effort. However, we aimed at diminishing this threat by adopting metrics already employed in prior software engineering literature (Çarka et al., 2022; Lavazza et al., 2025), while leaving room for future work to implement our methodology with further developer effort metrics.

Internal Validity. Internal validity is the extent to which an experimental design accurately identifies a cause-and-effect relationship between variables (Wohlin et al., 2012). Our study relies on a large-scale analysis of 365 OSS projects contained in the PANDORA dataset, which can potentially be biased from the project selection. We address this threat by designing our inclusion criteria based on past studies Palomba et al. (2018); Sellitto et al. (2022). Moreover, given the absence of prior studies on the long-term, i.e., ripple effects of refactoring, we have not formulated a specific hypothesis regarding potential confounding factors. Our exploratory study will consider identifying these factors throughout the research process. Therefore, we acknowledge that this approach may introduce unforeseen confounders, which we will address as they arise and include in the exploratory analysis. We acknowledge the existing threat to validity in considering *regular expression* matching within the commit messages for the issue-fixing commit detection. Hence, we acknowledge the use of this technique as a best effort given its complexity, and it is currently a hot topic in Software Engineering research (Esposito and Falessi, 2023). Moreover, given the relatively low specificity (TNR ≈ 0.49) obtained from the manual validation implying a non-trivial rate of false positives, we acknowledge this method to overestimate defect-proneness to some extent, i.e. labelling non bug-fix commits as fixing. Nevertheless, the high score in Recall ensured these cases to represent a minor group. We also recognize that other latent factors, such as developer expertise, project-specific guidelines, or concurrent maintenance activities, may influence the evolution of classes and the impact of refactorings. This choice reflects a broader trade-off commonly accepted in large-scale software repository mining studies, where the breadth of data analyzed across heterogeneous projects enhances ecological validity but limits the control over all potential confounding factors (Palomba et al., 2018; Spadini et al., 2018). Instead of introducing artificial constraints to enforce isolation, we prioritized maximizing realism and generalizability of findings, as also recommended in empirical software engineering guidelines Runeson and Höst (2009).

External Validity. External validity concerns how the research elements (subjects, artifacts, etc.) represent actual elements (Wohlin et al., 2012). Mining versioning or project management systems, particularly GitHub, threaten external validity. We mitigate this threat by utilizing our previously published data, PANDORA (Nguyen et al., 2022), which has already addressed and mitigated this same threat. We successfully mined and analyzed a considerable sample size of refactoring cases, yet we did not reach to analyze all the mined refactorings due to excessive computational time. We acknowledge this as a minor threat, given that the sample size analyzed already allows for fair generalizability considering the magnitude of analyzed refactorings and unique refactoring type. Similarly, the dataset focuses on Java-based open-source systems, which may limit the generalizability of our findings to other ecosystems such as industrial, closed-source, or non-Java-based projects. However, we can claim pontential to still reach generalizability beyond Java projects, since, if further mining tools like RefactoringMiner would exist for other programming languages, our approach could perfectly be employed with projects based on other programming languages. We further acknowledge that the use of tools like RefactoringMiner for detecting refactorings and regular expressions for issue-fixing commit extraction introduces a level of approximation. While these tools are widely validated and accepted in the literature, they may introduce false positives or negatives. Nevertheless, we have followed the previous research methodology to mitigate such threats, and consequently leading this issue to be a minor threat to validity.

Conclusion Validity. Statistical tests threaten the conclusion's validity regarding the appropriateness of statistical tests and procedures, such as assumption violation, multiple comparisons, and Type I or Type II errors. We address this issue using WT instead of the t-test due to the rejection of the normal data distributions. Moreover, we applied the Bonferroni correction(Kim and Choi, 2021), to reach a good balance between type 1 and type 2 errors by setting our alpha to 0.01, hence reducing it from the standard value of 0.05 due to the many statistical tests we performed. Spearman's ρ is a non-parametric measure of the strength and direction of a correlation between two variables, but due to multiple factors, this approach may hinder the conclusion's validity. For instance, the correlation coefficient is unreliable with a small sample size and cannot accurately represent the relationship between variables. Moreover, Spearman assumes a linear monotonic correlation between the variables; if the assumption were to prove invalid or not applicable, Spearman's correlation might not accurately measure the strength and direction of the association. We have mitigated this issue by ensuring a representative sample and checking the data distribution. We could mitigate this issue by using other non-parametric measures of correlation. Still, our analysis found that Spearman's ρ was more suitable than other correlation measures, such as Kendall's τ, because it can effectively handle tied observations in the data. On the other hand, Kendall's τ relies on concordant and discordant pairs of observations. The performed LCS algorithm alternative approach for the RE approximation provides line-by-line comparisons between two versions of the analyzed class source code. However, it might not understand word-level changes, for instance. We concluded to use this approach to not to deviate from the original code survival definition focusing on the *share of lines remaining unchanged.*

8 Related Work

This section reports our study's existing related work, including the current state of research on refactoring activity, as well as on the application of the RE on topics within the field of Software Engineering. We provide a detailed comparison with out work in Table 12.

It is common for individuals without formal training in software engineering to develop software (Wahler et al., 2016). Therefore, unstructured or documented or poorly designed code leads to hard-to-maintain software projects (Wahler et al., 2016). For instance, Wahler et al. (Wahler et al., 2016) presented a case study where software engineers assisted magnetics researchers in improving their codebase by leveraging refactoring supported by static analysis and software metrics. Thus, they studied the impact of their refactoring activity through single-snapshot observations on faults, code duplicates, and changes in lines of code. Refactoring software can enhance various dimensions of software quality, including readability and maintainability (Nyamawe et al., 2018). Nyamawe et al. (Nyamawe et al., 2018) studied this aspect by analyzing the impact of refactoring activity on traceability entropy and, further, validating these results through a qualitative voting from professional developers. Nonetheless, refactoring is also a time and resource-consuming task and, as such, should be carefully exploited as Kurbatova et al. (Kurbatova et al., 2020) demonstrated by leveraging ML classification on a sample of 14 software projects.

Refactoring is a behavior-preserving source code transformation (Abid et al., 2022). As such, albeit preventing alteration to the behavior of software projects, it impacts both the development process and the product quality (Abid et al., 2022; Cedrim et al., 2017b; Sellitto et al., 2022). Refactoring is currently commonly classified into 10 unique types of code transformation (Cedrim et al., 2017b). Cedrim et al. (Cedrim et al., 2017b) investigated how commonly used refactoring types impact the presence of code smells over time in 23 projects. Despite the majority of refactorings affecting smelly elements, only a small percentage reduces occurrences of smells, with a significant portion introducing new ones. Refactoring tends to introduce long-living smells rather than eliminate existing ones. They identified typical refactoring-smell patterns, such as the Move Method and Pull Up Method, inducing the emergence of the God Class and Extract Superclass, creating Speculative Generality in many cases. Sellitto et al. (Sellitto et al., 2022) investigated software refactoring impacts on program comprehension, particularly focusing on program readability. Their study, conducted on 156 open-source projects, highlighted the impact of refactoring on various program comprehension metrics. In the same vein, Lin et al. (Lin et al., 2019) investigated whether refactoring activities can enhance the naturalness of code. For that, they concentrated on analysing a set of 10 unique refactoring types to inspect their impact on code naturalness. After conducting an investigation involving 619 projects, the authors revealed that refactoring does not always lead to increased naturalness in the code and underscores the significant influence of refactoring types on code naturalness.

Furthermore, Traini et al. (Traini et al., 2021) investigated the impact of refactoring on software performance beyond its effects on maintainability. Their study involved the analysis of a set of 16 unique refactoring types over a set of 17 projects, whose impact was inspected via metrics such as project execution time, performance change, as well as the magnitude of the latter one. The authors revealed that refactoring can significantly influence execution time, with none of the investigated refactoring types ensuring no performance regression. Particularly, those aimed at decomposing complex code entities, like Extract Class/Interface or Extract Method, pose higher risks of performance degradation. Similarly, Rachatasumri et al. (Rachatasumrit and Kim, 2012) investigated the impact of 20 refactoring-type edits on regression tests using the version history of 3 Java OSS projects.

Table 12: Related work on the existing research on Refactoring Analysis.

Related work	**Sample Size**	**Refactoring**				**Time Dependence**
		#types	**Analysis**	**RE**	**Impact**	
Wahler et al. (2016)	1 project	undefined	Subjective developer feedback, Quantitative analysis from SATs	No	Faults (FindBugs), Code duplicates (PMD), LOC	Single Snapshot
Nyamawe et al. (2018)	85 code smells* (1 project)	2	Quantitative and qualitative evaluation on refactoring recommendations	No	Traceability entropy Entity Placement Developers' voting	Single Snapshot
Kurbatova et al. (2020)	14 projects	1	ML classification Quantitative evaluation	No	Precision, Recall, F1	Single Snapshot
Abid et al. (2022)	30 projects 15 developers	15	Correlation analysis Qualitative survey	No	QMOOD metrics Code Security Metrics Correlation coefficients Developers' perception	Single Snapshot
Cedrim et al. (2017b)	23 projects 16,566 refactorings	10	Longitudinal multinomial classification of refactoring impacts on code smells	No	Code Smells	Single Snapshot
Sellitto et al. (2022)	156 projects 96,268 commits	79	Multinomial Log-Linear Regression	No	Code Readability	Single Snapshot
Lin et al. (2019)	619 projects 1,448 refactorings	10	Statistical Inference (Wilcoxon signed-rank test)	No	Cross-Entropy	Double Snapshot (Before and after)
Traini et al. (2021)	17 projects 69 commits 1,534 refactorings	16	Correlation analysis Qualitative manual inspection Statistical comparison of refactoring density	No	Execution Time Performance Change Magnitude of Performance Change	Double Snapshot (Before and after)
Rachatasumrit and Kim (2012)	3 projects 1676 commits 1113 refactorings	20	Quantitative descriptive analysis	No	RTC, CTC, TTC, ATR, ACR, ACRF, RTCT	Yes (Inter-release analysis)
Reddy and Rao (2009)	4 hypothetical example designs	0	Definition of dependency oriented complexity metrics	Yes	DOCMS(R), DOCMA(CR), DOCMA(AR)	Single Snapshot
Robbes and Lungu (2011)	2463 projects 2942 contributors	2	Heuristic-based method deprecation detection Qualitative manual inspection	Yes	Addition of provider RE, Removal of provider RE, Rename of provider RE	Ripple propagation over time
Our work	48 projects 4,766 commits 108,523 refactorings	96	Statistical Inference (Wilcoxon signed-rank test, Dunn's All-Pairs test) Correlation Analysis (Spearman's Correlation Coefficient)	Yes	Refactoring impact RE, CP, DP Change Efficiency	Ripple propagation over time

Analyzing refactoring reconstruction and change impact, the authors explored the relationship between refactoring types/locations and affected changes/tests.

Recently, another study tackled the definition of the "Ripple Effect". For instance, Reddy et al. (Reddy and Rao, 2009) investigate an RE in the context of design changes. The authors focus on detection techniques for design defects, including shotgun surgery and divergent change, using probability-based metrics computed on components and their dependencies; it considers deltas between different design points. Their definition of RE is developed around the propagation of design defects, affecting multiple components and increasing system complexity. Similarly, Robbes et al. (Robbes and Lungu, 2011) focuses on a RE in the form of API changes and, more specifically, on the propagation of changes across a software ecosystem, their impact over time computed on entire software systems, and their interdependencies. The authors create a meta-model to save each version's

deltas with the immediate predecessor. The RE is defined, in this context, as the addition or removal of software entities following the four types of effects the authors have identified.
Conversely, our definition of the RE considers refactoring over time from a continuous point of view. Therefore, we consider how long a given refactoring action influences the code quality of software entities over the entire system development time. Hence, we do not consider discrete differences, i.e., the delta between pairs of successive versions. Instead, we analyze them continuously to detect the long-term effect of single refactoring. Therefore, considering different types of refactoring, our scope of study is broader. Hence, the novelty of our contributions lies in the pioneering of a new methodology to analyze code quality with time as a variable without making it discrete. Finally, Abid et al. (Abid et al., 2022) investigated the process of developers deciding whether to apply recommended refactorings, highlighting the challenge of assessing their impact on the system. They introduce RIPE (Refactoring Impact PrEdiction), a technique that estimates the effects of refactoring operations on source code quality metrics. They highlighted that evaluating the pros and cons of refactoring is far from trivial.

9 Conclusions

Our study provided a comprehensive analysis of the long-term impact of refactoring through the lens of the Ripple Effect (RE), examining its persistence, influence on change and defect proneness, and benefit-to-effort trade-offs across 98 refactoring types. Our findings reveal that while most REs persist for fewer than 10 subsequent changes, certain structural refactorings can endure across more than 190, with statistically significant differences across refactoring families. Long-lasting REs, especially those in Split and Invert families, are more prone to defects and effort-intensive modifications, whereas mid-term REs (lasting 9–19 changes) are more stable and efficient.

We show that the temporal span of a refactoring's influence is a decisive factor in its sustainability: medium-lived refactorings offer the best trade-off between stability and cost-effectiveness, while overly persistent ones can deteriorate maintainability. Interestingly, certain families like Assert consistently exhibit strong stability, low change-proneness, and favorable benefit-to-effort ratios.

Overall, our work not only offers new empirical evidence on how refactoring impacts code evolution but also delivers actionable guidelines: **not all persistent refactorings are valuable, i.e., those that persist just long enough without overreaching tend to yield the most sustainable outcomes**. In future work, we aim to investigate the implications of RE on broader software quality metrics and reevaluate past research findings in light of persistent ripple effects.

Declarations

Author Contributions: We specify our contributions according to the CRediT taxonomy:

– **Mikel Robredo:** Writing – Original Draft Preparation, Data Curation, Software, Data Curation. Mikel provided essential data preparation and contributed to drafting specific sections and contributed to writing the registered report.

– **Matteo Esposito:** Methodology, Writing – Original Draft Preparation. Matteo developed the primary research methodology and contributed significantly to writing the initial manuscript draft and registered report.
– **Fabio Palomba** Methodology, Validation, Writing – Review & Editing, Supervision. Fabio critically validated the study's results, supervised the research's methodological approach, and reviewed the manuscript for academic rigor.
– **Rafael Peñaloza:** Formal Analysis, Supervision. Rafael performed a thorough formal data analysis and statistical techniques and guided the research through supervision.
– **Valentina Lenarduzzi:** Conceptualization, Methodology, Supervision, Validation, Writing – Review & Editing. Valentina led the conceptualization and methodological framing of the research, supervised the entire project, validated findings, and contributed to reviewing and refining the manuscript.

Equal Author Contribution: Mikel Robredo and Matteo Esposito contributed equally to this work.
Conflict of Interest: We declare that we have no competing interests.
Data Availability Statement: We provide our raw data and analysis results in our replication package hosted on Zenodo[13].
Ethical Approval: Our work did not need ethical approval.
Funding: This work has been funded by FAST, the Finnish Software Engineering Doctoral Research Network, funded by the Ministry of Education and Culture, Finland.
Original Registered Report: We provide our original registered report on SSRN [14].

References

Abid C, Kessentini M, Alizadeh V, Dhaouadi M, Kazman R (2020) How does refactoring impact security when improving quality? a security-aware refactoring approach. IEEE Transactions on Software Engineering 48(3):864–878

Abid C, Kessentini M, Alizadeh V, Dhaouadi M, Kazman R (2022) How does refactoring impact security when improving quality? a security-aware refactoring approach. IEEE Transactions on Software Engineering 48(3):864–878

Alikhanifard P, Tsantalis N (2025) A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. ACM Transactions on Software Engineering and Methodology 34(2), DOI 10.1145/3696002, URL https://doi.org/10.1145/3696002

Amoroso d'Aragona D, Bakhtin A, Li X, Su R, Adams L, Aponte E, Boyle F, Boyle P, Koerner R, Lee J, et al. (2024) A dataset of microservices-based open-source projects. In: Proceedings of the 21st International Conference on Mining Software Repositories, pp 504–509

Anderson TW, Darling DA (1952) Asymptotic Theory of Certain "Goodness of Fit" Criteria Based on Stochastic Processes. The Annals of Mathematical Statistics 23(2):193 – 212

Arvanitou EM, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P (2017) A method for assessing class change proneness. In: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, pp 186–195

Avgeriou PC, Taibi D, Ampatzoglou A, Fontana FA, Besker T, Chatzigeorgiou A, Lenarduzzi V, Martini A, Moschou A, Pigazzini I, et al. (2020) An overview and comparison of technical debt measurement tools. Ieee software 38(3):61–71

Barsade SG (2002) The ripple effect: Emotional contagion and its influence on group behavior. Administrative science quarterly 47(4):644–675

[13] https://doi.org/10.5281/zenodo.15209117
[14] https://papers.ssrn.com/sol3/Delivery.cfm?abstractid=4938000

Basili VR, Caldiera G, Rombach HD (1994) The goal question metric approach. Encyclopedia of Software Engineering

Bauer DF (1972) Constructing confidence sets using rank statistics. Journal of the American Statistical Association 67(339):687–690

Billingsley P (2013) Convergence of probability measures. John Wiley & Sons

Çarka J, Esposito M, Falessi D (2022) On effort-aware metrics for defect prediction. Empirical Software Engineering 27(6):152

Casella G, Berger RL (2002) Statistical inference duxbury press. Pacific Grove, CA[Google Scholar]

Cedrim D, Garcia A, Mongiovi M, Gheyi R, Sousa L, De Mello R, Fonseca B, Ribeiro M, Chávez A (2017a) 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, pp 465–475

Cedrim D, Garcia A, Mongiovi M, Gheyi R, Sousa L, de Mello R, Fonseca B, Ribeiro M, Chávez A (2017b) Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects. In: Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, p 465–475

Chowdhury RA, Ramachandran V (2006) Cache-oblivious dynamic programming. In: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, Citeseer, pp 591–600

Chvatal V, Sankoff D (1975) Longest common subsequences of two random sequences. Journal of applied probability 12(2):306–315

Cohen I, Huang Y, Chen J, Benesty J, Benesty J, Chen J, Huang Y, Cohen I (2009) Pearson correlation coefficient. Noise reduction in speech processing pp 1–4

Conover WJ (1999) Practical nonparametric statistics. john wiley & sons

D'Agostino RB (1986) Goodness-of-fit-techniques, vol 68. CRC press

Davis C, Stephens MA (1977) The covariance matrix of normal order statistics. Communications in Statistics-Simulation and Computation 6(1):75–81

Denaro G, Pezze M (2002) An empirical evaluation of fault-proneness models. In: Proceedings of the 24th International Conference on Software Engineering, pp 241–251

Dolgui A, Ivanov D, Sokolov B (2018) Ripple effect in the supply chain: an analysis and recent literature. International journal of production research 56(1-2):414–430

Dunn OJ (1961) Multiple comparisons among means. Journal of the American statistical association 56(293):52–64

Dunn OJ (1964) Multiple comparisons using rank sums. Technometrics 6(3):241–252

Esposito M, Falessi D (2023) Uncovering the hidden risks: The importance of predicting bugginess in untouched methods. In: 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, pp 277–282

Esposito M, Moreschini S, Lenarduzzi V, Hästbacka D, Falessi D (2023) Can We Trust the Default Vulnerabilities Severity? In: Moonen L, Newman CD, Gorla A (eds) 23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023, IEEE, pp 265–270, DOI 10.1109/SCAM59687.2023.00037, URL `https://doi.org/10.1109/SCAM59687.2023.00037`

Falessi D, Laureani SM, Çarka J, Esposito M, Costa DAd (2023) Enhancing the defectiveness prediction of methods and classes via jit. Empirical Software Engineering 28(2):37

Feynman RP (1963) The Feynman lectures on physics, vol 1. Addison-Wesley

Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional

Fowler M, Beck K (1999) Refactoring: Improving the design of existing code. Addison-Wesley Longman Publishing Co, Inc

Gelman A, Carlin JB, Stern HS, Rubin DB (1995) Bayesian data analysis. Chapman and Hall/CRC

Gill J (2002) Bayesian methods: A social and behavioral sciences approach. Chapman and Hall/CRC

Haney FM (1972) Module connection analysis: a tool for scheduling software debugging activities. In: Proceedings of the December 5-7, 1972, fall joint computer conference, part I, pp 173–179

Hoijtink H (2009) Bayesian data analysis

Hollander M, Wolfe DA, Chicken E (2013) Nonparametric statistical methods. John Wiley & Sons

Hunt JW, MacIlroy MD (1976) An algorithm for differential file comparison. Bell Laboratories Murray Hill

Iannone E, Codabux Z, Lenarduzzi V, De Lucia A, Palomba F (2023) Rubbing salt in the wound? a large-scale investigation into the effects of refactoring on security. Empirical Software Engineering 28(4):89

IEEE (1990) Ieee standard glossary of software engineering terminology. IEEE Std 61012-1990 pp 1–84

Kim JH, Choi I (2021) Choosing the level of significance: A decision-theoretic approach. Abacus 57(1):27–71

Kim M, Zimmermann T, Nagappan N (2012) A field study of refactoring challenges and benefits. In: International Symposium on the Foundations of Software Engineering

Kim M, Zimmermann T, Nagappan N (2014) An empirical study of refactoringchallenges and benefits at microsoft. IEEE Transactions on Software Engineering 40(7):633–649

Kurbatova Z, Veselov I, Golubev Y, Bryksin T (2020) Recommendation of move method refactoring using path-based representation of code. In: International Conference on Software Engineering Workshops, p 315–322

Lacerda G, Petrillo F, Pimenta M, Guéhéneuc YG (2020) Code smells and refactoring: A tertiary systematic review of challenges and observations. Journal of Systems and Software 167

Lavazza L, Morasca S, Rotoloni G (2025) Software defect prediction evaluation: New metrics based on the roc curve. Information and Software Technology p 107865

Lee PM (1997) Bayesian Statistics: An Introduction. Arnold, London

Lenarduzzi V, Saarimäki N, Taibi D (2020) Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study. Journal of Systems and Software 170:110750, DOI https://doi.org/10.1016/j.jss.2020.110750

Lenarduzzi V, Besker T, Taibi D, Martini A, Fontana FA (2021) A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. Journal of Systems and Software 171:110827

Li X, Zhang H, Le VH, Chen P (2024) Logshrink: Effective log compression by leveraging commonality and variability of log data. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp 1–12

Lima DL, Santos RDS, Garcia GP, Da Silva SS, França C, Capretz LF (2023) Software testing and code refactoring: A survey with practitioners. In: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 500–507

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

Lu H, Zhou Y, Xu B, Leung H, Chen L (2012) The ability of object-oriented metrics to predict change-proneness: a meta-analysis. Empirical software engineering 17:200–242

Mishra P, Pandey CM, Singh U, Gupta A, Sahu C, Keshri A (2019) Descriptive statistics and normality tests for statistical data. Annals of cardiac anaesthesia 22(1):67–72

Montgomery DC, Runger GC (2020) Applied statistics and probability for engineers. John wiley & sons

Mooij AJ, Ketema J, Klusener S, Schuts M (2020) Reducing code complexity through code refactoring and model-based rejuvenation. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, pp 617–621

Nguyen H, Lomio F, Pecorelli F, Lenarduzzi V (2022) PANDORA: Continuous mining software repository and dataset generation. In: EEE International Conference on Software Analysis, Evolution and Reengineering (SANER2022), IEEE

Nyamawe AS, Liu H, Niu Z, Wang W, Niu N (2018) Recommending refactoring solutions based on traceability and code metrics. IEEE Access 6:49460–49475

Ó Cinnéide M, Yamashita A, Counsell S (2016) Measuring refactoring benefits: a survey of the evidence. In: Proceedings of the 1st International Workshop on Software Refactoring, pp 9–12

Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2018) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng 23(3):1188–1221

Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2019) Toward a smell-aware bug prediction model. IEEE Transactions on Software Engineering 45(2):194–218

Peruma A, Simmons S, AlOmar EA, Newman CD, Mkaouer MW, Ouni A (2022) How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. Empirical Software Engineering 27(1):11

Rachatasumrit N, Kim M (2012) An empirical investigation into the impact of refactoring on regression testing. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp 357–366

Razali NM, Wah YB, et al. (2011) Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. Journal of statistical modeling and analytics 2(1):21–33

Reddy KR, Rao AA (2009) Dependency oriented complexity metrics to detect rippling related design defects. ACM SIGSOFT Software Engineering Notes 34(4):1–7

Robbes R, Lungu M (2011) A study of ripple effects in software ecosystems (nier track). In: Proceedings of the 33rd International Conference on Software Engineering, pp 904–907

Royston P (1992) Approximating the shapiro-wilk w-test for non-normality. Statistics and computing 2:117–119

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14(2):131–164

Saarimäki N, Robredo M, Lenarduzzi V, Vegas S, Juristo N, Taibi D (2025) Does microservice adoption impact the velocity? a cohort study. Empirical Software Engineering 30(5):130

Sandelowski M (1995) Sample size in qualitative research. Research in nursing & health 18(2):179–183

Sellitto G, Iannone E, Codabux Z, Lenarduzzi V, De Lucia A, Palomba F, Ferrucci F (2022) Toward understanding the impact of refactoring on program comprehension. In: International Conference on Software Analysis, Evolution and Reengineering, pp 731–742

Shaphiro S, Wilk M (1965) An analysis of variance test for normality. Biometrika 52(3):591–611

Silva D, Silva J, Santos GJDS, Terra R, Valente MTO (2020) Refdiff 2.0: A multi-language refactoring detection tool. IEEE Transactions on Software Engineering

Soares E, Ribeiro M, Gheyi R, Amaral G, Santos A (2022) Refactoring test smells with junit 5: Why should developers keep up-to-date? IEEE Transactions on Software Engineering 49(3):1152–1170

Soares V, et al (2020) On the relation between complexity, explicitness, effectiveness of refactorings and non-functional concerns. In: Brazilian Symposium on Software Engineering, pp 788–797

Song Q, Jia Z, Shepperd M, Ying S, Liu J (2011) A general software defect-proneness prediction framework. IEEE Transactions on Software Engineering 37(3):356–370

Spadini D, Aniche M, Bacchelli A (2018) Pydriller: Python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 908–911

Spearman C (1961) The proof and measurement of association between two things. The American Journal of Psychology

Sprinthall RC (2011) Basic statistical analysis. Pearson Education

Stephens MA (1974) Edf statistics for goodness of fit and some comparisons. Journal of the American Statistical Association 69(347):730–737

Stephens MA (2017) Tests based on edf statistics. In: Goodness-of-fit-techniques, Routledge, pp 97–194

Szóke G, Antal G, Nagy C, Ferenc R, Gyimóthy T (2014) Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp 95–104

Tempero E, Gorschek T, Angelis L (2017) Barriers to refactoring. Communications of the ACM 60(10):54–61

Traini L, Di Pompeo D, Tucci M, Lin B, Scalabrino S, Bavota G, Lanza M, Oliveto R, Cortellessa V (2021) How software refactoring impacts execution time. ACM Trans Softw Eng Methodol 31(2)

Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history. In: International Conference on Software Engineering, pp 483–494

Wahler M, Drofenik U, Snipes W (2016) Improving code maintainability: A case study on the impact of refactoring. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 493–501

Wilcoxon F (1945) Individual comparisons by ranking methods. Biometrics 1(6):80

Wohlin C, Runeson P, H0st M, Ohlsson M, ¨and A Wesslen BR (2000) Experimentation in Software Engineering: An Introduction. Springer

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B (2012) Experimentation in Software Engineering. Springer

Yau S, Collofello J, MacGregor T (1978) Ripple effect analysis of software maintenance. In: The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78., pp 60–65

Appendix

In this appendix, we describe additional content to leverage by the reader to better understand the insights of the descriptions provided in the main text of this study.

A Mitigation of False Positives in Bug Fixing Commits

In order to manually validate the accuracy of the regular expression-based approach adopted to identify bug fixing commits, we designed an experiment in which manually validate the automatically performed bug-fixing commit classifications. To such end, we created three different samples of a size of 383 commit observations, thus yielding a 95% confidence level and a 5% margin of error from the obtained results of the manual validation to be performed (Billingsley, 2013; Sandelowski, 1995). Furthermore, we performed additional resampling with replacement in those cases in which the initial random sampling added duplicated commit observations within the same sample, as well as across the three samples generated. Thus, we ensured the uniqueness of the observations being used in the manual annotation.

Subsequently, we examined the commit messages of the sampled commits, and classified them as bug-fixing following the same criteria defined when using regex pattern recognition. Within this procedure, two of the authors performed the manual annotation across the three drawn samples independently. In case of disagreement among the two manual annotators, a third author was tasked to independently review the disagreement cases in order to build the ground-truth. Following existing research, we evaluated the deviation of the leveraged regex issue ticket pattern recognition from the ground truth by assessing their classification output through a set of classification accuracy (Falessi et al., 2023). Table 13 presents the average results of the performed experiments with the defined three different samples. Similarly, we made the spreadsheet used to perform the manual annotations as well as the codes to perform the described sampling available in the replication package, shared within the submitted manuscript (see Data Availability Statement).

Table 13: Average results from the performed mitigation experiment.

Metric	Result	Metric	Result
Accuracy	0.8572	F1-Score	0.9101
Precision	0.8358	Matthews Correlation Coefficient (MCC)	0.6344
Recall (TPR)	0.9988	False Positive Rate (FPR)	0.5129
Specificity (TNR)	0.4870	False Negative Rate (FNR)	0.0011
Inter-Annotator Agreement	0.9617		

The average classification accuracy across the three samples was 85.73%, indicating a high overall correctness in labeling issue-fixing commits. Similarly, the average recall (TPR) of 99.88% suggested that the adopted approach was highly effective at detecting actual issue-fixing commits. This evidence highly supports our efforts to mitigate the potential existence of false positives within the analyzed observations. Even in the presence of some false positives results for F1-Score averaged 91.01% and a precision of 83.59% demonstrated that the pattern recognition approach we used is reliable in classifying bug-fixing commits. However, an obtained specificity (TNR) of 48.71% indicated that our approach did not perform optimally to identify non-bug-fixing commits, and hence it was aligned with the resulting false positive rate (FPR) of 51.29%. We also computed the Matthews Correlation Coefficient (MCC), which assesses the correlation between predicted and true classifications. The resulting MCC score of 0.63 suggested a moderate correlation, which supports our claim for adopting the presented pattern recognition approach. Lastly, we wanted to assess the alignment among the employed human annotators, which were key to identify the ground truth and thus compare the predicted bug-fixing commits with the true ones. The Inter-Annotator Agreement resulted in an averaged 96.17% confirming a high degree of labeling consistency across human annotators. This evidence ensures the validity of the employed ground-truth labellings.

Overall, the results suggested an acceptable accuracy on the used bug-fixing labelling classification approach. However, we also observed an unavoidable pattern to label bug-fixing as actually normal commits, which should be taken into caution in future works.

B Statistical Tests

In this section of the appendix, we describe the mathematical definitions of the statistical tests used in our study.

B.1 The Anderson-Darling test

To statistically test the normality on the distribution of the analyzed data, we used the AD normality test as explained in Section 3.4. This test leverages the cumulative distribution function of the data to calculate the test statistic A to compare it against the critical values of the theoretical distribution, in our use case the *Gaussian* normal distribution Stephens (1974). Hence, the value of the test statistic A is estimated as follows:

$$A = -n - \frac{1}{n}\sum_{i=1}^{n}[2i-1][ln(p_{(i)} + ln(1 - p_{(n-i+1)})] \tag{10}$$

where $p_{(i)} = \Phi([x_{(i)} - \overline{x}]/s)$. Here, Φ is the cumulative distribution function of the standard normal distribution, and $\overline{x}$ and s are the mean and standard deviation of the data values. The p-value is computed from the modified statistic $Z = A(1.0 + 0.75/n + 2.25/n^2)$ according to Table 4.9 in Stephens (1974). Further description on how the resulting *p-values* are computed from the estimated A test statistic can be found in Section 4.8.2 from D'Agostino (1986).

B.2 The Shapiro-Wilk test:

A more robust statistical test than the AD test, yet restricted to smaller sample size Razali et al. (2011), is the SW normality test, which offers effective estimated results for small to moderate-sized sample sizes (ideally $X \leq 50$, but up to 2000) Royston (1992). Thus, the SW test tests the null hypothesis that a sample's distribution follows normality through the W test statistic as follows:

$$W = \frac{R^4\hat{\sigma}^2}{C^2S^2} = \frac{b^2}{S^2} = \frac{(a'y)^2}{S^2} = \frac{(\sum_{i=1}^{n} a_i x(i))^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{11}$$

where:

$$\begin{aligned} R^2 &= m'V^{-1}m, \\ C^2 &= m'V^{-1}V^{-1}m, \\ a' = (a_1, \cdots, a_n) &= \frac{m'V^{-1}}{(m'V^{-1}V^{-1}m)^{\frac{1}{2}}}, \\ b &= R^2\hat{\sigma}/C \end{aligned}$$

Thus:

- C is the normalized constant based on a vector norm Davis and Stephens (1977).
- V us the covariance matrix of the normal order statistics Davis and Stephens (1977).
- m is a vector made from the expected values of the standard normal order statistics Davis and Stephens (1977),

- thus making b the best linear unbiased estimate of the slope of a linear regression of the ordered sample observations x_i.
- and a_i being the coefficient of the *i-th* normal order statistic Shaphiro and Wilk (1965).
- $x_(i)$ is the *i-th* normal order statistic Davis and Stephens (1977).
- $\bar{x}$ is the sample mean.

The formulation of the null (H_o) and alternative (H_a) hypotheses are defined in Section 3.4.

B.3 The Wilcoxon Signed-Rank test:

Given the scenario of a non-normal distribution of the data, the WT test stands as a nonparametric statistical alternative to test whether the distribution of two populations are equal, or otherwise unequal, considering the case of the *two-sided* use case of this test Hollander et al. (2013). Given one of the sample populations, the test statistic T is computed as follows:

- The observed values are sorted $|X_1|, \cdots, |X_n|$, and subsequent ranks are assigned based on each sorted position consequently $|R_1|, \cdots, |R_n|$.
- The test statistic is then calculated based on the signed-rank sum:

$$T^+ = \sum_{1 \leq i < n, X_i > 0} R_i, \tag{12}$$

$$T^- = \sum_{1 \leq i < n, X_i < 0} R_i \tag{13}$$

- Thus, W test statistic is then defined as $min(T^+, T^-)$.
- Given the fact that the sample size for our study is higher than 25, the normal approximation is assumed via the *z-value* Bauer (1972):

$$z = \frac{W - \mu_W}{\sigma_W} \tag{14}$$

where the mean (μ) is computed as:

$$\mu_w = \frac{n(n+1)}{4} \tag{15}$$

and the standard deviation (σ_W) is computed as:

$$\sigma_W = \sqrt{\frac{n(n+1)(2n+1) - \sum \frac{t_i^3 - t_i}{2}}{n-1}} \tag{16}$$

(n is the sample size and t is the number of times the i^{-th} value occurs)

Thus, the significance of the test statistic is determined by computing the p-value of the standard normal distribution Hollander et al. (2013) to test the hypothesis definitions described in Section 3.4.

B.4 The Dunn's test (with Bonferroni correction):

Given a setting in which two or more samples are compared to test whether they come from identically distributed distributions, the Dunn's test stands as a non-parametric approach to test multiple comparisons using rank sums Dunn (1964). Considering a set of sample comparisons p, the test calculates the values of the contrasts as follows:

$$y_m = \sum_i \frac{T_i}{\sum_i n_i} - \sum_{i'} \frac{T_{i'}}{\sum_{i'} n_{i'}} \tag{17}$$

where $m = 1, \cdots, p$, and the summations over i and i' are over distinct subsets of the integers $1, \cdots, k$. Similarly, the standard deviation of each of the comparisons is computed, and we provide the formulas of both the scenarios where ties are found and when they are not, since we automated this calculation:

$$\sigma_m = \begin{cases} \sqrt{[\frac{N(N+1)}{12}][(\sum_i n_i)^{-1} + (\sum_{i'} n'_i)^{-1}]} & \text{if ties do not exist,} \\ \sqrt{[\frac{N(N+1)}{12} - \frac{\sum_{s=1}^{r}(t_s^3 - t_s)}{12(N-1)}]_{(\frac{1}{\sum_i n_i} + \frac{1}{\sum_{i'} n'_i})}} & \text{otherwise.} \end{cases} \tag{18}$$

where *p-values* are then computed as $y_1/\sigma_1, \cdots, y_p/\sigma_p$. Given the chosen critical value α in Section 3.4, each of the computed *p-values* is compared with $z_{1-\alpha/2p}$, $1 - \alpha/2p$ being based on the standard normal distribution. Subsequently, and given that in this study we perform the *Bonferroni* adjustment Dunn (1961), the *p-values* are adjusted as $p' = p \times m$ where m is the number of comparisons being made as explained before.

B.5 The Spearman's ρ correlation coefficient:

The Spearman's ρ correlation coefficient stands as a rank-based re-definition of the *Pearson correlation coefficient* Conover (1999). Hence, for a sample size n, the obtained scores (e.g. (X_i, Y_i)) are translated into ranks $R[X_i], R[Y_i]$, which then they are used to compute ρ as follows:

$$\rho[R[X], R[Y]] = \frac{cov[R[X], R[Y]]}{\sigma_{R[X]}\sigma_{R[Y]}} \tag{19}$$

where the numerator is the covariance of the rank variables from each group, and the denominator is composed by the product of the standard deviations of the rank variables Spearman (1961). The interpretation of the obtained ρ coefficient can be described as:

$$\begin{cases} 0 < \rho \leq 1 & \text{indicates a positive direction of association,} \\ -1 \leq \rho < 0 & \text{indicates a negative direction of association.} \end{cases} \tag{20}$$

C BCP Threshold Selection

In this study, we chose as threshold the probability of $P(X) = 0.01$ to declare when the impact of the refactoring RE was negligible in the codebase. To assess the robustness of this choice, we conducted a sensitivity analysis by applying alternative thresholds. Since there are no previous studies addressing the long-term impact of refactoring through the analysis of its RE, we chose a set of alternative thresholds starting from the threshold used in this study ($P(X) = 0.01$), and ranging until a probability $P(X) = 0.20$. We considered more than 20% of the originally introduced code remaining in the codebase to be enough to declare that the refactored code at R_0 still had impact on the affected Java class.

We measured the survival of the initially identified REs in terms of number of observed REs, that is, the number of refactoring operations that, given the new threshold, still maintained subsequent refactorings of the same type implemented within the same Java class. Similarly, we measured the longevity of the identified REs in terms of average day duration of the RE, as well as in terms of average number of refactoring commits (see Table 14).

The results from the sensitivity analysis indicated that the number of refactoring REs remains unchanged across the chosen thresholds, with only a marginal reduction from 3,422 cases at probabilities of 0.01 and 0.05, to 3,419 at 0.10, and 3,417 at 0.20. Similarly, the average longevity measured in days decreased only from 3,316 days at thresholds 0.01 and 0.05 to 3,260 days at 0.20. In terms of commits, the average longevity dropped from 31.09 at 0.01 to 29.30 at 0.20. These results demonstrated that while stricter thresholds (e.g., 0.01) capture marginally longer REs, the overall patterns and conclusions remain consistent regardless of the specific threshold chosen, to some extent. Thus, while our threshold selection does not follow an existing convention in the refactoring literature, our sensitivity analysis demonstrated that the findings are not biased by the specific threshold chosen, but rather remain stable across reasonable variations.

Table 14: RE longevity metrics across varying thresholds

Threshold	Observed REs	Mean RE longevity (Days)	Mean RE longevity (# Commits)
0.01	3,422	3,316.56	31.10
0.05	3,422	3,316.56	31.04
0.10	3,419	3,316.54	30.72
0.20	3,417	3,259.91	29.30

D On the BCP's performance with data scarcity

We present the theoretical background and a collected refactoring case example to depict the issue encountered during the study when leveraging BCP for approximating the refactoring RE when there was data scarcity. Therefore, we first introduce the theoretical background to demonstrate in more detail the application of the Bayesian probability updating theory on our use case. Then, we present an example refactoring case scenario of a refactoring type collected during the study, implement BCP as described in this appendix, and discuss the potential risks of considering such results given the scenario of data scarcity.

D.1 Theoretical background

Given the nature of the problem, the probability distribution of the random variable, i.e. *share of lines remaining unchanged* should be defined on the basis of an interval as of $[0, 1]$. Therefore, the prior distribution shall be a *Beta* distribution (Gelman et al., 1995), which is characterized as a suitable probability distribution to analyze the random behaviour of percentages and proportions and, therefore, applicable for probabilities as well. The *Beta* distribution is defined as:

$$\begin{cases} \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1} & if x \in R_X \\ 0 & if x \notin R_X \end{cases} \tag{21}$$

where:

- X is a continuous random variable.
- B is the Beta function defined as:

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} \tag{22}$$

  where Γ denotes the gamma function (Casella and Berger, 2002).
- α and β are the shape parameters of the distribution.
- the mean and variance of the random variable distribution are:

$$\mu = \frac{\alpha}{\alpha+\beta} \tag{23}$$

$$\sigma^2 = \frac{\alpha\beta}{(\alpha+\beta+1)(\alpha+\beta)^2} \tag{24}$$

As the prior belief, we relied on the definition of code refactoring, which aims at improving the design, readability, maintainability and long-term viability of software systems (Szóke et al., 2014). Therefore, a strong prior should emphasize a strong belief that the code will remain stable in the future due to the performed refactoring. For instance, we introduced such strong prior in the mean of the *Beta* distribution as 99% probability for the code remaining unchanged in the concerning Java class. Consequently, the posterior mean probability should update with every subsequent refactoring case happening within the same Java class and categorized with the same refactoring type, until the would not be more instances following the mentioned criteria, or the RE would be inexistent based on the criteria defined in Section 3.4.

D.2 Refactoring case example

We now consider a refactoring case from the collected data to visualize the scenario of data scarcity. Specifically, consider the refactoring case details presented in Table D.1 for refactoring type *Rename Parameter* from project *apache/jmeter*. As performed with all the refactoring cases analyzed throughout the study, the class changes were computed in all the subsequent commits in which the same refactoring type was applied within the same affected Java class.

Table D.1: Refactoring case details from collected *Rename Parameter* refactoring case example.

Refactoring type	Rename Parameter
Project	https://github.com/apache/jmeter
Affected class	src\core\org\apache\jmeter\gui\util\TextAreaCellRenderer.java
Refactoring commit	c2ec8a122aeffd52b4207c8dba381619b5c709e7
Subsequent refactoring commits	69cb24419be822f605e56f0960d60eea57eb44aa, b7799e1602f257ab491c836bd57f416c87c5c04e, 17ae7f8cb9f128c5a37d0858deaf29039011a514, f6dafdf60d669a815ae677f9a87fcafd28c6197c

Considering the same prior belief setting defined in the theoretical example (as such belief should also apply in reality), we computed the share of changed lines and of unchanged lines in each subsequent commit. Table D.2 shows the number of changed and unchanged lines within the specified subsequent commits. We then computed the BCP, thus obtaining the posterior mean probability of the shared lines remaining unchanged. Figure D.1 depicts the update progression of the posterior mean probability during the detected subsequent commits along with the 95% *Confidence Interval* (CI), thus demonstrating the range of values the BCP model estimates the probabilities could most likely have.

If we were not conscious of the importance of having robust enough historical data supporting a high level of stability, the results depicted in Figure D.1 could lead us to consider that the BCP estimates the long durability of the source code on the observed Java class. However, making such a statement with a sample window of four observations should be interpreted with caution since a change like the one occurring in the second commit within the series could have reduced the probability more if the prior probability had been set lower. This evidence shows the powerful impact a strong prior assumption can have over a drastic probability update when few data points exist to update such initial prior probability.

Bayesian inference relies on the importance of making prior probability assumptions based on domain knowledge of the data distribution and similarly relies on the probabilistic update of the distribution based on future evidence (Gelman et al., 1995). Therefore, although Bayesian inference relies more on the CI of the performed estimations than on the sample size required in frequentist probabilistic approaches (Gelman et al., 1995), this example demonstrates the

Table D.2: Change history for refactoring case example in terms of **nloc** from *Jmeter* project. (**: Refactoring happened in parallel to a *rename class* refactoring)

Refactoring commit	**Unchanged lines**	**Changed lines**	**nloc**
69cb24419be822f605e56f0960d60eea57eb44aa	25	1	26
b7799e1602f257ab491c836bd57f416c87c5c04e	5	31	36
17ae7f8cb9f128c5a37d0858deaf29039011a514	36	3	39
f6dafdf60d669a815ae677f9a87fcafd28c6197c	39	0**	39

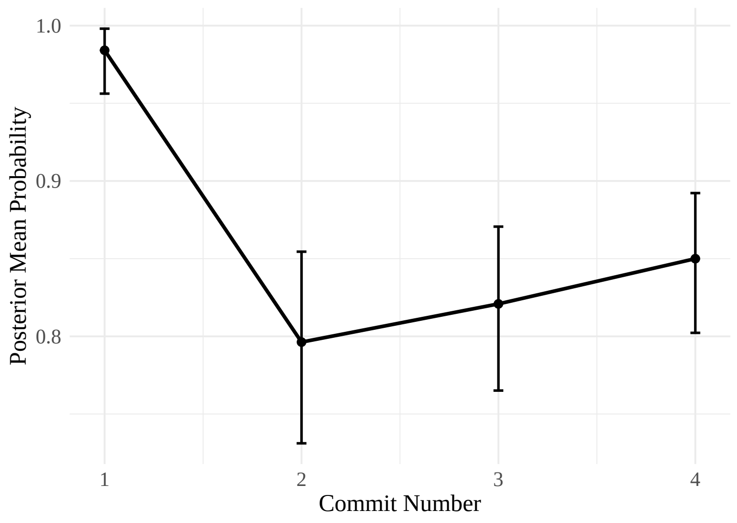

Fig. D.1: Bayesian mean posterior probability update.

pitfalls of facing data scarcity while assuming an optimistic (and yet theoretically correct) prior probability.

E List of Abbreviations from Table 12

Table E.1: List of abbreviations for the concepts included in Table 12 for the Related Works

Abbreviation	**Extended name**
LOC	Lines of Code
QMOOD	Quality model for object-oriented design
R	Set of methods and fields that are part of refactoring edits
C	Set of methods and fields that are part of atomic changes
T	Set of methods and fields exercised by existing tests
ATR	Affecting tests that exercise at least one refactoring edit location
ACR	Affecting changes whose location overlaps with at least one refactoring edit
ACRF	Subset of affecting changes for the failed tests that exercise the location of refactoring edits
RT	Intersection between R and T
CT	Intersection between C and T
DOCMS(R)	Dependency oriented complexity metric for structure
DOCMA(CR)	Dependency oriented complexity metric for an artifact causing ripples
DOCMA(AR)	Dependency oriented complexity metric for an artifact affected by ripples
RE	Ripple Effect
CP	Change Proneness
DP	Defect Proneness

F Additional Results

Table F.2: Dunn's Test Results on the Difference of RE Duration between RF (RQ$_1$ - H_{02})

Family A	Family B	ΔScore	Std Err Dif	Z	p-Value	Family A	Family B	ΔScore	Std Err Dif	Z	p-Value
Invert	Assert	37080.6	2645.822	14.0148	<.0001	Split	Parameterize	2519.3	2153.743	1.1697	1
Split	Assert	34935.4	2692.558	12.9748	<.0001	Try	Remove	2452.5	2646.263	0.9268	1
Parameterize	Assert	32416.1	2027.342	15.9895	<.0001	Rename	Localize	2179	2255.851	0.966	1
Reorder	Assert	31629.2	3482.674	9.0819	<.0001	Extract	Encapsulate	2101	1413.322	1.4865	1
Extract	Assert	28436.1	1854.895	15.3303	<.0001	Rename	Move	2034.8	449.093	4.531	0.0011
Inline	Assert	27358.9	1969.65	13.8902	<.0001	Try	Replace	1837.8	2721.991	0.6752	1
Rename	Assert	26650.6	1850.952	14.3983	<.0001	Modify	Localize	1816.3	2289.68	0.7933	1
Encapsulate	Assert	26335.1	2297.436	11.4628	<.0001	Inline	Encapsulate	1023.7	1560.886	0.6559	1
Modify	Assert	26287.8	1892.034	13.8939	<.0001	Replace	Remove	614.7	730.755	0.8412	1
Split	Push	24782.4	3179.917	7.7934	<.0001	Try	Merge	606.1	2830.31	0.2142	1
Move	Assert	24615.7	1870.035	13.1632	<.0001	Rename	Modify	362.7	533.335	0.6802	1
Localize	Assert	24471.5	2895.55	8.4514	<.0001	Rename	Encapsulate	315.5	1408.143	0.224	1
Invert	Add	24331.7	1918.479	12.6828	<.0001	Move	Localize	144.2	2271.535	0.0635	1
Split	Pull	23047	2155.518	10.6921	<.0001	Modify	Encapsulate	-47.3	1461.724	-0.0324	1
Try	Assert	22208.4	3209.308	6.92	<.0001	Rename	Inline	-708.3	764.313	-0.9267	1
Split	Add	22186.5	1982.437	11.1915	<.0001	Reorder	Parameterize	-786.9	3085.088	-0.2551	1
Merge	Assert	21602.3	2105.32	10.2608	<.0001	Pull	Add	-860.5	893.282	-0.9633	1
Reorder	Push	21476.2	3871.824	5.5468	<.0001	Modify	Inline	-1071	859.037	-1.2468	1
Replace	Assert	20370.6	1957.282	10.4076	<.0001	Inline	Extract	-1077.2	773.813	-1.3921	1
Invert	Change	19988.9	1919.138	10.4156	<.0001	Replace	Merge	-1231.6	1241.754	-0.9919	1
Remove	Assert	19755.9	1850.519	10.6759	<.0001	Move	Modify	-1672.1	596.201	-2.8046	0.9573
Reorder	Pull	19740.7	3086.327	6.3962	<.0001	Move	Encapsulate	-1719.4	1433.135	-1.1997	1
Parameterize	Add	19667.2	888.991	22.1231	<.0001	Push	Pull	-1735.5	2641.925	-0.6569	1
Reorder	Add	18880.2	2968.031	6.3612	<.0001	Rename	Extract	-1785.5	381.166	-4.6843	0.0005
Split	Change	17843.7	1983.075	8.998	<.0001	Remove	Merge	-1846.4	1065.562	-1.7328	1
Change	Assert	17091.7	1845.018	9.2637	<.0001	Localize	Encapsulate	-1863.6	2634.588	-0.7074	1
Rename	Push	16497.6	2507.594	6.5791	<.0001	Split	Invert	-2145.2	2743.878	-0.7818	1
Extract	Add	15687.2	347.598	45.1301	<.0001	Modify	Extract	-2148.3	546.863	-3.9283	0.0163
Parameterize	Change	15324.4	890.413	17.2105	<.0001	Try	Localize	-2263.1	3458.684	-0.6543	1
Split	Remove	15179.5	1988.194	7.6348	<.0001	Try	Move	-2407.3	2659.947	-0.905	1
Rename	Pull	14762.1	906.871	16.2781	<.0001	Push	Add	-2596	2502.711	-1.0373	1
Inline	Add	14609.9	748.138	19.5284	<.0001	Move	Inline	-2743.1	809.434	-3.3889	0.1333
Split	Replace	14564.8	2087.929	6.9757	<.0001	Merge	Localize	-2869.3	2468.848	-1.1622	1
Reorder	Change	14537.5	2968.458	4.8973	0.0002	Localize	Inline	-2887.3	2354.222	-1.2264	1
Rename	Add	13901.6	325.902	42.6559	<.0001	Move	Extract	-3820.3	465.077	-8.2144	<.0001
Encapsulate	Add	13586.2	1399.43	9.7084	<.0001	Localize	Extract	-3964.6	2259.087	-1.7549	1
Modify	Add	13538.9	509.886	26.5528	<.0001	Try	Modify	-4079.4	2675.459	-1.5248	1
Split	Merge	13333.1	2227.3	5.9862	<.0001	Replace	Localize	-4100.9	2343.884	-1.7496	1
Try	Push	12055.4	3627.9	3.323	0.1692	Try	Encapsulate	-4126.7	2975.987	-1.3867	1
Pull	Assert	11888.4	2029.227	5.8586	<.0001	Replace	Move	-4245.1	778.854	-5.4505	<.0001
Reorder	Remove	11873.3	2971.88	3.9952	0.0123	Try	Rename	-4442.2	2646.565	-1.6785	1
Move	Add	11866.8	420.977	28.1888	<.0001	Parameterize	Invert	-4664.5	2095.022	-2.2265	1
Localize	Add	11722.6	2250.422	5.2091	<.0001	Remove	Localize	-4715.6	2255.496	-2.0907	1
Extract	Change	11344.4	351.22	32.2999	<.0001	Merge	Encapsulate	-4732.8	1728.934	-2.7374	1
Parameterize	Merge	10813.9	1349.487	8.0133	<.0001	Remove	Move	-4859.8	447.306	-10.8647	<.0001
Invert	Encapsulate	10745.5	2357.375	4.5582	0.001	Try	Inline	-5150.5	2730.898	-1.886	1
Split	Localize	10463.9	2985.416	3.505	0.0868	Pull	Change	-5203.3	894.697	-5.8157	<.0001
Try	Pull	10320	2774.175	3.72	0.0378	Reorder	Invert	-5451.4	3522.502	-1.5476	1
Split	Move	10319.6	2006.372	5.1434	<.0001	Merge	Inline	-5756.6	1261.159	-4.5645	0.001
Inline	Change	10267.2	749.828	13.6927	<.0001	Rename	Parameterize	-5765.6	902.645	-6.3874	<.0001
Replace	Push	10217.7	2587.075	3.9495	0.0149	Replace	Modify	-5917.2	830.286	-7.1267	<.0001
Push	Assert	10153	3095.703	3.2797	0.1974	Replace	Encapsulate	-5964.5	1545.249	-3.8599	0.0216
Reorder	Merge	10026.9	3136.881	3.1965	0.2643	Try	Extract	-6227.7	2649.324	-2.3507	1
Invert	Inline	9721.7	2039.246	4.7673	0.0004	Replace	Rename	-6280	731.85	-8.5809	<.0001
Remove	Push	9602.9	2507.274	3.83	0.0243	Remove	Modify	-6531.9	531.832	-12.2819	<.0001
Rename	Change	9558.9	329.763	28.9872	<.0001	Remove	Encapsulate	-6579.2	1407.574	-4.6742	0.0006
Try	Add	9459.5	2641.94	3.5805	0.0652	Merge	Extract	-6833.8	1073.143	-6.368	<.0001
Encapsulate	Change	9243.4	1400.334	6.6009	<.0001	Push	Change	-6938.7	2503.217	-2.7719	1
Modify	Change	9196.1	512.362	17.9485	<.0001	Replace	Inline	-6988.2	994.537	-7.0266	<.0001
Merge	Add	8853.3	1054.78	8.3935	<.0001	Remove	Inline	-7603	763.264	-9.9611	<.0001
Split	Modify	8647.6	2026.892	4.2664	0.0038	Replace	Extract	-8065.5	741.766	-10.8733	<.0001
Invert	Extract	8644.5	1928.636	4.4822	0.0014	Remove	Extract	-8680.2	379.059	-22.8993	<.0001
Split	Encapsulate	8600.3	2409.711	3.569	0.0681	Try	Reorder	-9420.8	3963.244	-2.377	1
Replace	Pull	8482.2	1107.856	7.6564	<.0001	Pull	Merge	-9713.8	1352.318	-7.1831	<.0001
Split	Rename	8284.8	1988.597	4.1662	0.0059	Try	Parameterize	-10207.7	2772.796	-3.6814	0.0441
Parameterize	Localize	7944.6	2402.698	3.3065	0.1795	Rename	Invert	-10430	1924.844	-5.4186	<.0001
Remove	Pull	7867.5	905.988	8.6839	<.0001	Modify	Invert	-10792.8	1964.382	-5.4942	<.0001
Parameterize	Move	7800.4	941.157	8.2881	<.0001	Replace	Reorder	-11258.5	3039.507	-3.7041	0.0403
Replace	Add	7621.7	714.941	10.6606	<.0001	Push	Merge	-11449.3	2700.81	-4.2392	0.0043
Split	Inline	7576.5	2099.528	3.6087	0.0585	Replace	Parameterize	-12045.5	1104.399	-10.9068	<.0001
Move	Change	7524.1	423.972	17.7466	<.0001	Move	Invert	-12464.8	1943.202	-6.4146	<.0001
Localize	Change	7379.8	2250.984	3.2785	0.1983	Pull	Localize	-12583.1	2404.288	-5.2336	<.0001
Reorder	Localize	7157.6	3713.739	1.9273	1	Localize	Invert	-12609.1	2943.333	-4.2839	0.0035
Reorder	Move	7013.4	2984.071	2.3503	1	Remove	Parameterize	-12660.2	901.757	-14.0395	<.0001
Remove	Add	7007	323.436	21.6642	<.0001	Try	Split	-12727	3290.616	-3.8677	0.0209
Rename	Remove	6894.7	359.268	19.1909	<.0001	Pull	Move	-12727.3	945.211	-13.4651	<.0001
Split	Extract	6499.3	1992.268	3.2623	0.21	Assert	Add	-12748.9	1844.332	-6.9125	<.0001
Parameterize	Modify	6128.3	984.144	6.227	<.0001	Push	Localize	-14318.6	3353.538	-4.2697	0.0037
Parameterize	Encapsulate	6081	1633.083	3.7236	0.0373	Pull	Modify	-14399.4	988.021	-14.574	<.0001
Reorder	Modify	5341.3	2997.907	1.7817	1	Pull	Encapsulate	-14446.7	1635.422	-8.8336	<.0001
Reorder	Encapsulate	5294.1	3268.924	1.6195	1	Push	Move	-14462.8	2521.713	-5.7353	<.0001
Try	Change	5116.7	2642.419	1.9364	1	Try	Invert	-14872.2	3252.485	-4.5726	0.0009
Parameterize	Inline	5057.3	1126.173	4.4907	0.0013	Pull	Inline	-15470.4	1129.564	-13.6959	<.0001
Rename	Merge	5048.3	1066.313	4.7344	0.0004	Merge	Invert	-15478.3	2170.569	-7.131	<.0001
Reorder	Rename	4978.6	2972.149	1.6751	1	Push	Modify	-16134.9	2538.07	-6.3571	<.0001
Modify	Merge	4685.6	1136.131	4.1241	0.0071	Push	Encapsulate	-16182.2	2853.107	-5.6718	<.0001
Merge	Change	4510.6	1055.979	4.2715	0.0037	Pull	Extract	-16547.7	914.892	-18.087	<.0001
Change	Add	4342.8	290.311	14.959	<.0001	Replace	Invert	-16710	2027.302	-8.2425	<.0001
Reorder	Inline	4270.3	3047.486	1.4013	1	Push	Inline	-17205.9	2596.445	-6.6267	<.0001
Parameterize	Extract	3980	910.703	4.3703	0.0024	Remove	Invert	-17324.7	1924.428	-9.0025	<.0001
Split	Reorder	3306.2	3557.74	0.9293	1	Push	Extract	-18283.1	2510.506	-7.2826	<.0001
Replace	Change	3278.9	716.709	4.575	0.0009	Pull	Parameterize	-20527.7	1227.399	-16.7246	<.0001
Reorder	Extract	3193.1	2974.607	1.0734	1	Push	Parameterize	-22263.2	2640.477	-8.4315	<.0001
Move	Merge	3013.5	1099.105	2.7418	1	Pull	Invert	-25192.2	2096.846	-12.0143	<.0001
Remove	Change	2664.2	327.325	8.1393	<.0001	Push	Invert	-26927.6	3140.442	-8.5745	<.0001