What Were You Thinking? An LLM-Driven Large-Scale Study of Refactoring Motivations in Open-Source Projects

MIKEL ROBREDO, University of Oulu, Finland
MATTEO ESPOSITO, University of Oulu, Finland
FABIO PALOMBA, University of Salerno, Italy
RAFAEL PEÑALOZA, University of Milano-Bicocca, Italy
VALENTINA LENARDUZZI, University of Oulu, Finland

Context. Code refactoring improves software quality without changing external behavior. Despite its advantages, its benefits are hindered by the considerable cost of time, resources, and continuous effort it demands.
Aim. Understanding why developers refactor, and which metrics capture these motivations, may support wider and more effective use of refactoring in practice.
Method. We performed a large-scale empirical study to analyze developers' refactoring activity, leveraging Large Language Models (LLMs) to identify underlying motivations from version control data, comparing our findings with previous motivations reported in the literature.
Results. LLMs matched human judgment in 80% of cases, but aligned with literature-based motivations in only 47%. They enriched 22% of motivations with more detailed rationale, often highlighting readability, clarity, and structural improvements. Most motivations were pragmatic, focused on simplification and maintainability. While metrics related to developer experience and code readability ranked highest, their correlation with motivation categories was weak.
Conclusions. We conclude that LLMs effectively capture surface-level motivations but struggle with architectural reasoning. Their value lies in providing localized explanations, which, when combined with software metrics, can form hybrid approaches. Such integration offers a promising path toward prioritizing refactoring more systematically and balancing short-term improvements with long-term architectural goals.

CCS Concepts: • **Computer systems organization** → *Maintainability and maintenance*; • **Software and its engineering**; • **Computing methodologies** → **Artificial intelligence**; • **Information systems** → *Open source software*; Data mining;

Additional Key Words and Phrases: Software maintenance, Code refactoring, Refactoring motivations, Large Language Models (LLMs), Empirical software engineering, Software metrics, Mining software repositories, Developer behavior, Recommendation systems, Software quality

ACM Reference Format:
Mikel Robredo, Matteo Esposito, Fabio Palomba, Rafael Peñaloza, and Valentina Lenarduzzi. 2026. What Were You Thinking? An LLM-Driven Large-Scale Study of Refactoring Motivations in Open-Source Projects. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (February 2026), 62 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Mikel Robredo, University of Oulu, Oulu, Finland, mikel.robredo@oulu.fi; Matteo Esposito, University of Oulu, Oulu, Finland, matteo.esposito@oulu.fi; Fabio Palomba, University of Salerno , Salerno, Italy, fpalomba@unisa.it; Rafael Peñaloza, University of Milano-Bicocca, Milano, Italy, rafael.penaloza@unimib.it; Valentina Lenarduzzi, University of Oulu, Oulu, Finland.

ACM 1049-331X/2026/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Maintaining codebases in the era of computing pervasiveness in daily tasks, from smart appliances to autonomous vehicles, is becoming increasingly challenging [23]. With years of development and increasing reduction of time-to-releases, developers can make poor design choices to meet deadlines, leading to complex and hard-to-maintain software, thus increasing subsequent maintenance effort and costs for software firms [43]. Code refactoring is one of the most well-known techniques to mitigate software complexity [31]. Refactoring aims to introduce code structure changes without altering external behavior, from code optimization to architecture and design patterns [62].

Developers restructure the code to improve its design, readability, and maintainability to ensure the long-term sustainability of software systems [38, 41, 84]. Thus, developers perceive it as a significant time and resource allocation cost due to its complexity and limitations [7, 38]. Therefore, with limited time-to-release windows, developers often limit resources spent on refactoring, hindering the quality and maintenance of the code [7]. However, reducing software complexity with minor code refactoring can yield little to no improvement, leading to decreased code quality. In contrast, bulk and more comprehensive refactorings have a more profound influence on code quality [41, 84].

In an era in which developers benefit from the natural language processing capabilities LLMs bring when performing code refactoring [37, 63, 75, 77], current research efforts go beyond understanding their capability on performing direct code refactoring operations [21, 48]. Exploring LLM reasoning capabilities based on diverse variants of source code as information context [32], as well as non-idiomatic approaches for enhancing code refactoring [97] have built the recent research path on exploiting the reasoning capabilities of LLMs in code refactoring tasks.

Building upon this, extending the study conducted by Silva et al. [79] on the confessions of GitHub contributors about their motivations leading them to perform refactoring operations, our work aims at **identifying the motivations leading developers to perform code refactoring by leveraging LLMs**, and similarly, compares the extracted motivations with those validated by the software engineering literature [60, 79]. For that, we focus on mining the refactoring activity of software projects, and providing the LLMs with the source code and version control context from the project at the stage a refactoring operation is identified, thus investigating the LLMs' capabilities to identify developers' motivations leading to performing refactoring operations. In addition, our study examines the extent to which software metrics (SMs) reflect the refactoring motivations (RMs) identified by the LLMs, thereby investigating whether quantitative project characteristics can serve as indicators of developers' willingness to perform code refactoring.

Thus, our work is based on Silva et al. [79], but also extends Pantiuchina et al. [60]. More precisely, and comparing our work with the latter, while Pantiuchina et al. [60] analyzed pull-requests (PRs) based on keyword lookup and the RefactoringMiner tool (RMT) [3] to detect refactoring-related commits, we introduce **a novel LLM-based approach capable of analyzing all refactoring commits in the project history**, thus overcoming the limitations of PR-based analyses, for instance, incomplete change histories and rejected PRs, among others. Additionally, while their study focused primarily on product-quality indicators (e.g., OOP complexity, static analysis warnings, and developer-related measures), our work focuses on **established product and process metrics** widely adopted in defect prediction and quality-assessment research [36, 60, 66]. Lastly, while the goal of Pantiuchina et al. [60] was to examine correlations between product quality, developer metrics, and the likelihood of refactoring within individual PRs, we aim at qualitatively and quantitatively investigating the underlying motivations driving developers to perform refactoring operations, and thus define **a catalogue of motivations**.

To the best of our knowledge, no previous study leveraged LLMs to identify developer motivations for performing refactoring operations by exploiting their reasoning capabilities to analyze developers' commit messages and source code, and extract the code RMs. Our novel approach, which departs from conventional techniques, promises to deliver the first developer-based RMs to effectively target the correct amount of refactoring effort toward the right target. Our approach benefits practitioners, translating into a cost reduction, prioritizing refactoring efforts, and researchers paving the way for future works in developer-based software maintenance.

We designed and conducted a large empirical study analyzing **114 Java projects** hosted on GitHub.[1] Using RMT, we detected and analyzed **13,725,139 refactoring activities** across a comprehensive list of **783,002 commit histories**. We then used advanced LLMs to interpret the motivations of developers expressed in version control history data, analyzing their correlations with a robust set of established product and process metrics. This systematic approach enabled us to quantify the effectiveness of these metrics in capturing developer motivations and contributed to the development of a refined catalogue for developer-centric refactoring recommendation systems. Our study contributions are as follows:

- We explore the LLM reasoning capabilities on identifying developers' motivation leading them to perform refactoring operations.
- We evaluate the alignment between LLM-generated refactoring motivations and motivations reported in Silva's work [79], using expert judgment and statistical agreement analysis.
- We categorize and quantify how LLMs enrich or extend existing motivations with more detailed or context-aware rationale.
- We perform a large-scale LLM-assisted open-coding of the identified refactoring motivations into 14 categories and analyze their distribution and dominant trends.
- We assess the information power and correlation of process and product SMs in explaining refactoring motivations using Random Forest, Extreme Gradient Boost, and statistical correlation tests.
- We discuss the implications for researchers and practitioners, outlining directions for LLM-guided refactoring recommendation systems.

The application of the undergone study revealed that LLMs agreed with human judgment in 80% of the analyzed cases. However, only 47% of the RMs identified by the LLMs fully aligned with those reported by the reference study [79]. Interestingly, in 22% of the cases, LLMs extended known motivations with richer explanations, highlighting concerns like readability, testability, or naming clarity that were often implicit in prior work. A majority of motivations, over 55%, centred around improving code clarity or reducing redundancy, revealing a predominantly pragmatic intent behind refactoring. While developer experience and readability-related metrics emerged as important factors in machine learning models, their direct correlation with motivation categories was statistically weak. Our study confirms the potential of LLMs to identify developer rationale in localized refactorings, while also exposing their limitations in capturing architectural intent, underscoring the need for hybrid systems that blend LLM reasoning with contextual project data. To facilitate the reading of our study, we include Table E.1 (see Appendix E) with the glossary summarizing the acronyms of the main concepts used across the manuscript.

Paper Structure. In Section 2, we present the study design, while Section 3 presents the results and Section 4 discusses them. Section 5 focuses on threats to the validity of our study. Section 6 discusses related work, and in Section 7, we draw the conclusions.

2 STUDY DESIGN

This section outlines the empirical study, including the study goal and research questions, the study context, the data collection methodology, and the data analysis approach. Our empirical

[1]https://github.com

study follows established guidelines defined by Wohlin et al. [92]. We publish the raw data in the replication package. Figure 1 provides a graphical description of the conducted study design.

2.1 Goal and Research Questions

The *goal* of this study is to artificially identify the motivations that drive developers to perform refactoring activities by leveraging the capabilities of LLMs to interpret source code and natural language, with the *purpose* of identifying the product and process metrics that can help understand the motivations pushing developers to perform refactoring. The *perspective* is that of researchers and practitioners seeking additional support to guide developers in performing the refactoring. The *context* is open-source Java projects.

Therefore, we derived the following research questions (RQs):

RQ$_1$

Can LLMs extract developers' refactoring motivation behind a refactoring operation?

RQ$_{1.1}$ *What is the impact of adding different information context variants into the prompt on the performance of LLMs for identifying refactoring motivations?*

RQ$_{1.2}$ *How accurately can LLMs extract developers' refactoring motivations behind a refactoring operation?*

RQ$_{1.3}$ *What are the limitations in the LLMs' performance when identifying refactoring motivations?*

In recent years, the use of LLMs to automate various processes within the field of software engineering (SE) has grown rapidly [29, 96]. SE activities such as automated code generation, bug detection, and code refactoring have already experienced research efforts on the involvement of LLMs within the development process [45]. Diving deeper into code refactoring operations, research efforts have been made on understanding LLMs' capabilities on performing direct code refactoring [21, 48], as well as proposing hybrid approaches for refactoring non-idiomatic code [97]. Interestingly, Gao et al. [32] investigated the deductive code reasoning capability of multi-agent LLM pipelines based on source code as information context. Building on existing research performed on investigating the motivations behind performing refactoring operations [60, 79], we aim to investigate whether LLMs can effectively extract developers' refactoring motivations when performing refactoring operations. Moreover, we define a multi-agent LLM decision pipeline and ascertain its accuracy, as well as identify the limitations between the employed LLMs. Similarly, we explore whether including different context variants into the prompt has a positive or negative impact on the LLMs' refactoring motivation identification capabilities.

To build upon this, an efficient approach to explore how far LLMs are from providing refactoring motivations close to the real human perception is investigating its alignment with the state-of-the-art [79]. Hence, we ask:

RQ$_2$

Do motivations for refactoring in past studies align with those found in software project change histories?

Silva et al. [79] proposed a list of motivations for refactoring efforts provided by professional developers on their authored refactoring operations, and encompassing over 12 different Refactoring Types (RTs) (see Table 7). Pantiuchina et al. [60] built upon Silva et al.'s motivations, focusing on quality metrics and static analysis warnings, extracting information from PRs. However, PRs represent developer interaction, but cannot represent the entire project change history [66, 94]. Moreover, PRs can be rejected; thus, it is also challenging to discern expert contributions among PRs accepted and not [95]. Hence, we investigate developer motivation, analyzing the single commits from the projects analyzed in the reference study [79].

Furthermore, developers' commits, PRs, and code comments may not be able to capture all the motivations pushing developers to refactor the codebase; hence, we ask the following:

RQ$_3$

Are there additional motivations driving the developers' willingness to perform refactoring?

Refactoring improves source code quality [62]; yet, quality improvements in readability, performance, safety, and security are only part of the overall picture [59]. For instance, in an industrial context, it is essential to assess refactoring opportunities in situations with limited resources regarding work allocation and time [85]. Silva et al. [79] succeeded in identifying 36 refactoring RMs at the time they ran their study. Therefore, we investigate whether external limitations or additional motivations influence refactoring efforts.

Finally, while our work investigates whether LLMs can identify refactoring motivations based on refactoring contexts, there is no evidence on the extent to which these motivations are reflected in projects' data. Similarly, code refactoring is expected to affect software quality aspects such as those captured through product and process metrics. If the identified motivations to perform refactoring operations are based on developers' perceptions of the situation of their project, we would expect software metrics (SMs) to act as indicators of such context. Therefore, to explore whether SMs reflect the projects' context data upon which we identify refactoring motivations, we ask:

RQ$_4$

To what extent can product and process metrics reflect the motivations driving the developers' willingness to perform refactoring?

Based on Pantiuchina et al. [60], a relationship exists between specific quality metrics and refactoring operations. However, their metric selection was limited to quality metrics and static analysis warnings. Building upon our main focus on identifying developers' refactoring motivations through the use of LLMs, we are interested in exploring whether product and process metrics, which are characterized to reflect a systems change through the development process [28, 61], can actually reflect the motivations driving the developers' willingness to perform refactoring. Therefore, we extended the selection from Pantiuchina et al. [60] with established product and process metrics [36, 65], as well as a larger body of data used in our study, given the time frame between Pantiuchina's work and the present study, thus broadening the candidate for a plausible relationship to RMs. From now on, we will refer to these metrics as SMs during the study to enhance the comprehensibility of our work.

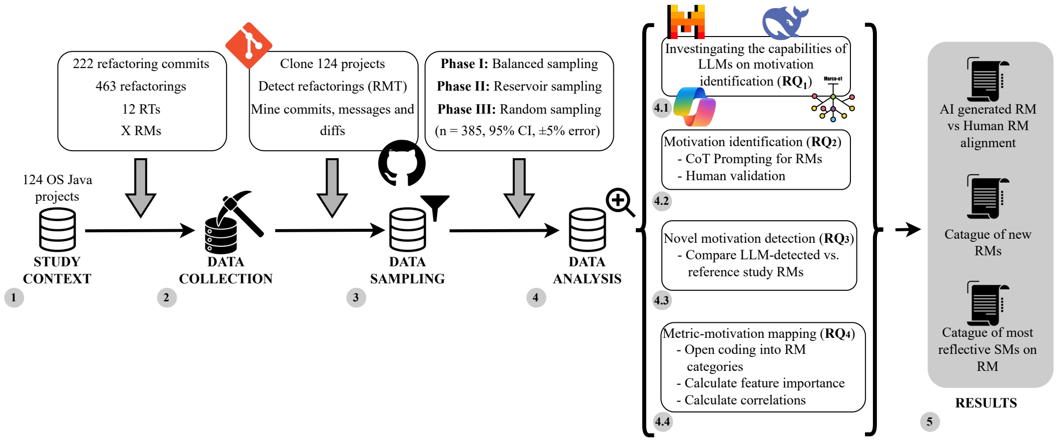

222 refactoring commits
463 refactorings
12 RTs
X RMs
Clone 124 projects
Detect refactorings (RMT)
Mine commits, messages and diffs
Phase I: Balanced sampling
Phase II: Reservoir sampling
Phase III: Random sampling
(n = 385, 95% CI, ±5% error)
124 OS Java projects
STUDY CONTEXT
1
DATA COLLECTION
2
DATA SAMPLING
3
DATA ANALYSIS
4
Investigating the capabilities of LLMs on motivation identification (RQ1)
4.1
Motivation identification (RQ2)
- CoT Prompting for RMs
- Human validation
4.2
Novel motivation detection (RQ3)
- Compare LLM-detected vs. reference study RMs
4.3
Metric-motivation mapping (RQ4)
- Open coding into RM categories
- Calculate feature importance
- Calculate correlations
4.4
AI generated RM vs Human RM alignment
Catague of new RMs
Catague of most reflective SMs on RM
RESULTS
5

Fig. 1. Workflow diagram of the study design (RT: Refactoring Type)

.

Therefore, we hypothesize that there must already be existing SMs in the commit history of software projects, highly correlated to the ground truth motivations and the hypothetically newly identified ones. The formulated hypotheses are defined as follows:

- $\mathbf{H}_{0C.1}$. There is no statistically significant correlation between the considered SMs and RMs.
- $\mathbf{H}_{1C.1}$. There is a statistically significant correlation between the considered SMs and RMs.

Consequently, considering H_{1C} as the accepted hypothesis, and therefore the evidence of correlation, we expand our hypotheses as follows:

- $\mathbf{H}_{0C.2}$. The identified correlation is negative.
- $\mathbf{H}_{1C.2}$. The identified correlation is positive.

In this sense, this question aims at identifying the metrics that can explain the existence of the stated motivations and further understand their specific implications. Therefore, we provide a list of SMs that better reflect the collection of RMs identified in our study.

2.2 Study Context

In this study, we adopted as the context dataset the one published by Silva et al. [79], which consists of a catalogue of 36 distinct motivations for 12 well-known refactoring types (RTs). In their work, they initially selected the top 1000 Java repositories in GitHub,[2] ordered in terms of popularity, and further filtered them, removing the lower quartile based on the number of commits, thus obtaining a list of 748 repositories with consistent maintenance activities. Additionally, they filtered out from the datasets such projects not showing development activity while conducting their study, thus reducing the number of projects to 471. The final set of considered projects comprised 124 repositories, which featured at least one refactoring during the study period, and answers provided by developers regarding the specific refactoring activity to the author's questions on the refactoring motivation. This set of projects comprised 222 commits, for which the authors provided answers for a total of 267 RMs and 36 unique motivations. Within this set of commits, 463 refactorings from 12 RTs were detected using the RMT [86]. Therefore, to validate the motivations provided by

[2]https://github.com

Table 1. Summary descriptive statistics of the final set of projects considered in the study

	Mean	Standard deviation	Median	Min	Max	Skew	Kurtosis	Standard error
Age	13.67	3.99	14.13	2.86	23.81	-0.36	0.18	0.38
Size	572,311.79	1,367,904.93	174,573	4,221	10,745,685	5.73	36.26	129,835.79
Stars	7,212.38	11,908.57	3,603	351	78,022	4.24	20.23	1,120.26
Open issues	398.24	655.53	177	0	4,278	3.51	14.52	62.79
Closed issues	4,064.06	5,871	2,445	55	38,493	3.94	18.60	562.34
Overall issues	4,462.29	6,253.89	2,912	60	40,400	3.82	17.76	599.01
Commits	21,231.53	47,849.40	7,818	344	401,429	5.73	38.41	4,501.29
Developers	347.50	451.02	195	11	2,305	2.53	6.49	42.43
Languages	4.42	3.54	4	1	22	2.37	7.70	0.33
Refactoring Commits	92.35	10.91	96	47	103	-1.93	3.83	1.03
Refactoring types	6,663.39	18,650.19	2,389	77	186,854	8.28	75.98	1,754.46
Refactorings	117,424.62	522,169.20	32,437	385	5,509,181	9.89	99.42	49,121.55

developers, our study considers the final set of 124 GitHub-hosted Java projects described in the adopted dataset.[3]

In Table 1, we present the descriptive statistics of the Java projects that completed all the stages in the data collection (see Section 2.3). We can observe that the population of repositories presents an average maturity of more than 13 years, with some projects indicating a maximum age of more than 23 years. Regarding the popularity of the studied projects, measured in terms of GitHub stars, we also observed a highly skewed distribution, with the most starred project holding 78,022 stars, while the mean remained at 7,212.38 and the median at 3,603. We also revealed extreme dispersion between projects based on characteristics such as the number of commits, which presented an average number of almost 21,000 commits, but with some projects containing as many as 401,000 commits. Similarly, the number of developers involved in a project varied from 11 to over 2,300, and the number of *secondary* programming languages used within the software repositories ranged from 1 to 22 (Java being their main programming language), thus depicting diverse levels of team size and project complexity. To compute the number of programming languages, we only considered programming languages following the criteria adopted from the TIOBE index.[4] According to these criteria, a programming language should have an entry in Wikipedia that defines it as a programming language, and further, it should have at least 5,000 results for Google search `<language> programming`. Furthermore, this index only considers programming languages that are *Turing complete* [83]. The TIOBE index contains 358 programming languages, which we provided in the replication package of this study (see Data Availability Statement).

Moreover, we computed the project's size based on effective lines of code (ELOC), which resulted in a highly skewed distribution of the projects (kurtosis = 36.26). Peak-sized projects presented almost 11 million ELOC, while the average size resulted in over 570,000 ELOC, which already denoted an average noticeable size of the considered projects. To compute ELOC, we quantified the total number by only considering the programming languages included in the TIOBE index. Altogether, we consider the study context for this study to be representative of an extended variety of Java projects in terms of the presented software attributes.

[3]http://aserg-ufmg.github.io/why-we-refactor
[4]https://www.tiobe.com/tiobe-index/programminglanguages_definition/#instances

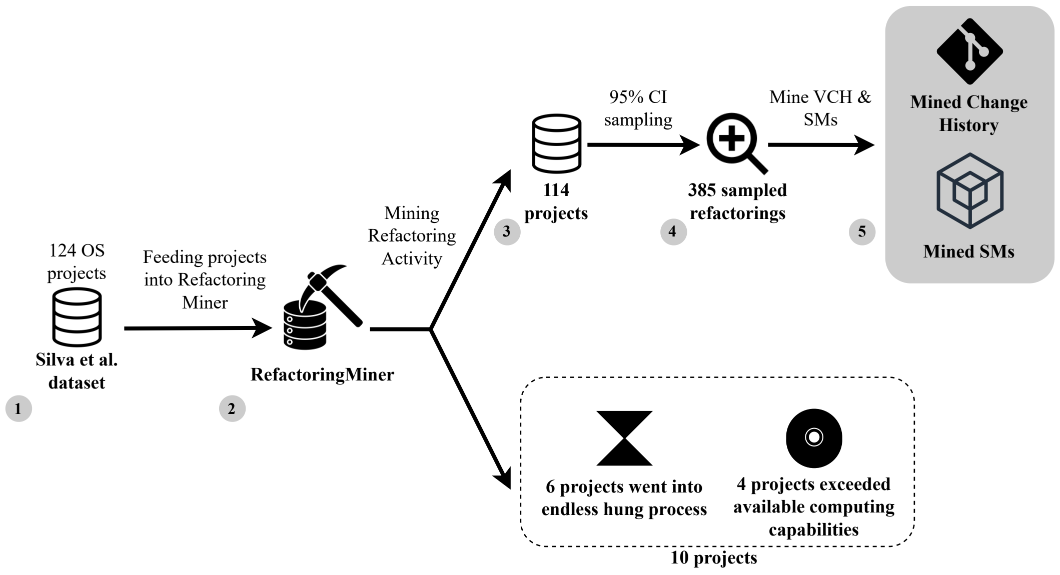


Fig. 2. Data collection process diagram (VCH: *Version Control History*, SM: *Software Metrics*).

2.3 Data Collection

This section outlines our multi-stage data collection process, depicted in Figure 2, and designed to support our RQs. The process includes: (i) mining refactoring data, (ii) creating a statistically significant sample for the analysis, (iii) mining project change history, and (iv) computing SMs listed in Table 4. To aid the narrative of our study design, we have included further details on the data collection process in Appendix A.

2.3.1 Collecting Refactoring Data. To ground our study in real-world developer activity, we mined the refactoring data from 124 GitHub projects, replicating and extending the dataset from the reference study [79]. Since their publication, these projects experienced an average increase of over 10,600 commits; e.g., *JetBrains/intellij-community* increased by 273,044 commits. We provide a table presenting the differences between the number of commits in the studied projects when the reference study was published and for the present study in the replication package (see Data Availability Statement).

We adopted the RefactoringMiner tool (RMT) [3] to mine the refactoring activity from the entire version-control history of the cloned repositories. Table 2 shows the categorization of the 103 refactorings of our study according to their types as defined by Fowler [31].

Due to resource limitations and persistent errors with RMT, we excluded 10 projects from the study. However, since we could still collect the SMs from affected refactoring commits, we included the affected commits and their ground-truth RMs. We acknowledge the impact of this decision as a threat to validity in Section 5. We extend the description of the data collection process and issues in Appendix A.1.

In total, we mined 114 projects, detecting 13,725,139 refactorings across 783,002 commits. Commits without refactorings were discarded. Abbreviations for refactoring types (RTs) used throughout the article are listed in Appendix Table C.1. Refactoring category frequencies are shown in Table 3. To expand the work conducted by Pantiuchina et al. [60], we also investigated the extent to which

Table 2. Refactorings Types detectable by RefactoringMiner 3.0 and newer versions [3].

RT Category	Description	Detectable Refactoring Type (RT)
Composing Methods	Reorganize 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 Classes	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 Class 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.
Syntax/Command 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.

(**Bold RTs**: Refers to the RTs investigated in the reference study (12 out of 103 RTs).)

Table 3. Mined frequencies for each RT detected by RefactoringMiner (#: Instances).

RT	#	RT	#	RT	#	RT	#	RT	#	RT	#
RM	1,997,099	MM	198,667	RC	76,450	RAA	35,118	TWR	8,283	SM	1,957
MMA	1,663,555	RCA	196,080	AAA	76,300	EC	34,097	EI	8,123	MerC	1,682
CVT	694,433	APA	182,867	ATET	73,858	IC	32,880	CTDK	8,117	MCat	1,677
AMA	649,691	RTET	181,235	RPM	69,931	CTET	28,877	AVA	7,783	CH	1,441
CPT	605,118	RA	171,885	CCAM	62,162	RAWV	28,764	RParam	7,739	RPWL	1,427
AP	504,837	IV	165,687	RAM	60,016	RCM	27,218	RAWC	5,173	SClass	1,016
CRT	438,291	ACA	144,488	IM	59,572	LP	22,840	MA	4,932	MerM	945
EV	360,499	CAAM	143,428	PV	55,221	MCode	22,184	AT	4,766	SP	877
MCA	342,591	AVM	133,253	RMM	55,131	EA	21,651	IA	4,752	MPack	705
CMAM	330,173	AAM	125,500	ACM	49,746	MCon	21,021	SP	4,372	SV	688
EM	324,173	RAWL	123,551	EnA	47,433	SC	18,579	ESub	4,187	RA	615
RenP	297,012	RVM	117,181	MARM	46,839	MSF	18,562	MARA	4,124	MVA	305
RV	292,827	RGWD	109,725	RVWA	46,643	Sup	16,199	MP	3,828	PT	73
CAT	282,403	PUM	102,417	RPA	43,379	PDA	15,160	RVA	3,745	-	-
MARM	281,148	APM	99,209	PUA	42,097	PA	11,683	MV	3,418	-	-
RP	261,442	MA	97,947	PDM	39,711	RCWT	10,976	RLWP	3,326	-	-
MovC	251,219	EAMM	88,375	MAIM	38,629	MParam	10,815	RPack	3,088	-	-
RMA	212,805	AMM	82,472	MAA	36,976	MPA	9,352	SA	2,622	-	-

refactorings occurred within pull requests (PRs) by mining the PR endpoint of their GitHub repository. Thus, among the collection of commits with detected refactoring activity, 25% were associated with PRs, while the remaining 75% represented commits performed outside PRs. Therefore, we expand the mined body of knowledge by over three times the prior research.

2.3.2 Selecting a Statistically Significant Sample for the Analysis. Analyzing more than 13 million refactorings was computationally infeasible due to the cost of SM extraction (e.g., ADEV, MINOR) (see Section 2.3.4) and prompt-based LLM analysis (see Section 2.4).

To ensure tractability and rigor, we selected a sample of 385 observations, yielding a 95% confidence level and 5% margin of error in the obtained results [10, 73]. For that, we followed a similar sampling approach to that of already implemented sampling techniques within the field of Software Engineering [44] (see Figure 3). We applied a greedy sampling strategy over three phases:

- **Phase 1**: Ensure each project and RT had a minimum of 3 sampled refactorings.
- **Phase 2**: Use reservoir sampling [88] to fill gaps in underrepresented projects.
- **Phase 3**: Apply random sampling to complete the remaining unfilled entries.

Our rationale for performing this sample strategy stands on prior work performed by Saarimäki et al. [70]. Among the major issues identified in their work within empirical SE research, they claimed the issue of representativeness of studied software projects when large projects dominate small ones. Our collected data consists of 114 software projects, in which software repositories such as *jetbrains/intellij-community* (almost half a million commits) and *apache/zookeeper* (2,700 commits) are similarly mined and included in our dataset. To remediate the highlighted issue in our

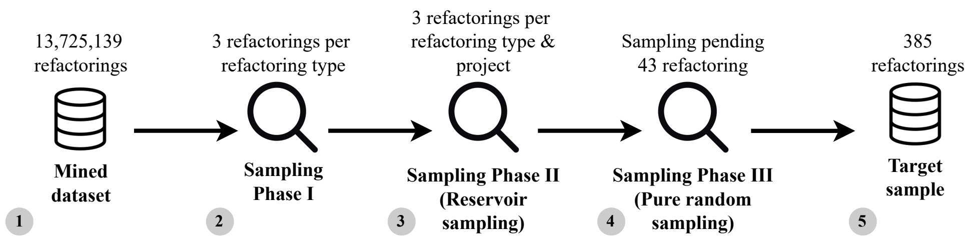

13,725,139 refactorings
Mined dataset
1
3 refactorings per refactoring type
Sampling Phase I
2
3 refactorings per refactoring type & project
Sampling Phase II (Reservoir sampling)
3
Sampling pending 43 refactoring
Sampling Phase III (Pure random sampling)
4
385 refactorings
Target sample
5

Fig. 3. Diagram of the adopted sampling strategy.

study, our sampling strategy aims at representing at least a minimal equal amount of refactoring types to research to analyze, and therefore represent, all the refactoring types mineable by RMT.

Since our dataset contains refactorings both inside and outside PRs, we verified that the drawn sample preserved a similar distribution. Following the same mining effort as in the data collection, we identified 34% of the sampled refactorings to be inside PRs, closely located to the 25% observed in the full dataset. This confirms that the sampling strategy does not introduce bias with respect to PR vs. non-PR activity and that the sample reliably allows us to generalize our results to the entire body of mined data. Nevertheless, we discuss the implications as threats to validity and potential further work on this within Section 5.

We expand on the description of the performed sampling strategy in Appendix A.2.

2.3.3 Mining the Change History. For each sampled commit, we used the PyDriller[5] Python library to extract the commit message and the corresponding file-level code diff where the refactoring was applied. PyDriller also supports basic structural analysis using Lizard,[6], enabling us to capture class- and method-level metrics across languages (e.g., Java, Python, C), thereby supporting the calculation of software metrics based on version control data.

2.3.4 Computing the SMs (Software Metrics). To answer RQ_3, we computed both product and process metrics [36, 65] (see Table 4) leveraging different tools. We used:

- **PyDriller** for SMs requiring the cumulative inspection of the version-control history between commits.
- **CK** [6] for product metrics [60] (e.g., WMC, RFC), invoked via CLI over target Java files.
- **CoRed**,[7] a Python wrapper of the official implementation of the metric described by Scalabrino et al. [74] to compute ComRead, which provides a comprehensive readability score.

Table 4 presents and describes all the metrics considered in this study.

2.4 Data Analysis

This section describes the data analysis techniques and methods used for answering our RQs, including our investigation of the capabilities of LLMs to extract developers' refactoring motivations (RQ_1), the identification of RMs for each of the sampled refactoring observations (RQ_2), the alignment and identification of additional RMs compared to the reference study (RQ_3), as well as the evaluation of the extent SMs reflect RMs within the software system (RQ_4). Figure 4 presents the workflow diagram of the data analysis.

[5]https://pydriller.readthedocs.io/en/latest/
[6]https://github.com/terryyin/lizard
[7]https://github.com/grosa1/CoRed/tree/master

Table 4. **Product** and **process** metrics adopted as our software metrics (SMs).

	Metric	Type	Description
Rahman and Devanbu [65]	ADD	Process	The normalized number of lines added to a given file in the considered commit.
	ADEV	Process	The cumulative number of active developers who modified a given file up to the considered commit.
	COMM	Process	The cumulative number of commits in a given file up to the considered commit.
	DELE	Process	The normalized number of lines removed from a given file in the considered commit.
	DDEV	Process	The cumulative number of distinct developers contributed to a given file up to the considered commit.
	EXP	Product	The mean of the experience of all developers across the project.
	MINOR	Product	The number of contributors who contributed less than 5% of a given file up to the considered commit.
	NADEV	Process	The number of active developers who changed any of the files involved in the commits where the given file has been modified.
	NCOMM	Process	The number of commits where the given file has been involved.
	NDDEV	Process	The number of distinct developers who changed any of the files involved in the commits where the given file has been modified.
	NSCTR	Process	The number of different packages touched by the developer in commits where the file has been modified.
	OEXP	Product	The percentage of code lines authored by a given developer in the project.
	OWN	Product	Measures the percentage of the lines authored by the highest contributor of a file.
	SCTR	Process	The number of packages modified by the committer in the considered commit.
Kamei et al. [36]	AGE	Process	The average period between the last and the current change.
	CEXP	Process	The number of commits performed on the given file by the committer up to the considered commit.
	ENTROPY	Product	The distribution of the modified code across each given file in the considered commit.
	FIX	Process	Whether or not the change is a defect fix.
	LA	Product	Ten lines added to the given file in the considered commit (absolute number of the ADD metric).
	LD	Product	The number of lines removed from the given file in the considered commit (absolute number of the DEL metric).
	LT	Product	The number of lines of code in the given file in the considered commit before the change.
	ND	Process	The number of directories involved in a commit.
	NDEV	Process	The number of developers that changed the modified files.
	NF	Process	Number of modified files.
	NS	Process	Number of modified subsystems (packages).
	NUC	Process	The number of times the file has been modified up to considered commit.
	REXP	Process	The number of commits performed on the given file by the committer in the last month.
	SEXP	Process	The number of commits a given developer performs in the considered package containing the given file.
Pantiuchina et al. [60]	CBO	Product	Coupling Between Object classes: measures the dependencies a class has.
	ComRead	Product	Comprehensive readability model combining structural, visual, and textual features.
	DIT	Product	Depth of Inheritance Tree.
	ELOC	Product	Effective Lines Of Code: code excluding blanks and comments.
	HsLCOM	Product	Henderson-Sellers LCOM: cohesion metric.
	NOC	Product	Number Of Children (direct subclasses).
	NOF	Product	Number Of Fields declared in a class.
	NOM	Product	Number Of Methods in a class.
	NOPF	Product	Number Of Public Fields declared in a class.
	NOPM	Product	Number Of Public Methods in a class.
	NOSF	Product	Number Of Static Fields declared in a class.
	NOSI	Product	Number Of Static Invocations of a class.
	NOSM	Product	Number Of Static Methods in a class.
	RFC	Product	Response For a Class: number of local and remote methods reachable.
	WMC	Product	Weighted Methods per Class: sums the cyclomatic complexity of the methods in a class.

2.4.1 LLM Roles, Human Validation and Prompting Strategy. Since the general availability of LLMs, our research community has investigated their use in SE tasks, benefiting both practitioners and researchers [24]. Among such tasks, classification and rating are the most time-consuming and error-prone tasks that humans perform during research activities [26, 27]. Therefore, since we performed LLM prompting in different stages of the data analysis to answer our RQs, we evaluated

Table 5. Overview of Selected LLMs

Model	Parameters & Quantization	Details	Highlights	Stochastic Control
Marco-o1 (LRM)[a]	7.6B, not quantized	Inspired by OpenAI's o-1	Fine-tuned on CoT datasets, uses MCTS + softmax scoring, excels at math, coding, and logic tasks	All models were executed using the same low-stochasticity decoding configuration: `temperature = 0.1`, `top_p = 0.9`, `top_k = -1`, `repetition_penalty = 1.0`, `max_new_tokens = 4096`. Under vLLM, a fixed seed was enforced for reproducibility, together with identical sampling parameters across all models.
Mistral-Nemo-Instruct-2407[b]	12.2B, not quantized	Alignment-tuned version of Mistral-Nemo-Base-2407	128K token context, strong instruction-following, competitive performance vs similar-size open models	
Phi-4 (Microsoft)[c]	14B, not quantized	Next-generation Phi series model	Strong reasoning, math, and coding; optimized for efficiency and safety	
DeepSeek-R1-Distill-Qwen-14B[d]	14B, not quantized	Distilled from DeepSeek-R1 using Qwen 14B architecture	Strong reasoning performance; competitive with much larger models; efficient inference	

[a] https://huggingface.co/AIDC-AI/Marco-o1

[b] https://huggingface.co/mistralai/Mistral-Nemo-Instruct-2407

[c] https://huggingface.co/microsoft/phi-4

[d] https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-14B

the proficiency, accuracy, and agreement of four LLMs assigned to extract the motivation behind refactoring operations. The following paragraphs describe the roles of the LLMs adopted in the data analysis design, the human validation employed in each of the response collections obtained from the LLMs, as well as the prompting strategy adopted to guide them in performing the task classifications. Table 5 presents the overview of the implemented setting for the experiments executed with selected LLMs, including characteristics such as model temperature or stochasticity level, among others.

LLM Roles. We employed four distilled LLMs for classification and validation tasks:

- **Large Reasoning Model (LRM):** Marco-o1 acted as the primary assistant for identifying RMs. Its output formed the basis for further validation and was referred to as the LRM response throughout the study.
- **First Validation Model (V1):** Mistral NeMo was assigned as the first validation assistant. It received the same input as the LRM and was additionally asked to evaluate the LRM's reasoning and output. Its role was to independently assess the correctness of the LRM's decision.
- **Second Validation Model (V2):** DeepSeek R1 acted as the second validation assistant, mirroring the role of V1. It provided a parallel evaluation of the LRM's output to detect agreement or disagreement with V1's assessment.
- **Third Validation Model (V3):** Microsoft Phi-4 was designated as the final arbiter in cases where V1 and V2 disagreed. In addition to the standard input, it received the assessments from both V1 and V2, as well as the original LRM output, to deliver a final decision in conflicting cases.

Each selected model was assigned a specific role based on its function in the data analysis pipeline (see Figure 4). We provide further technical details on the adopted LLMs, as well as model configuration in Appendix B.1.

Human Validation. Since no previous study reported findings on the accuracy of LLMs for the task at hand, we designed a validation strategy involving three human experts replicating the LLM validators' roles. Collectively, the experts tasked to perform the human validation bring more than two decades of hands-on industrial experience with mission-critical software systems, extensive

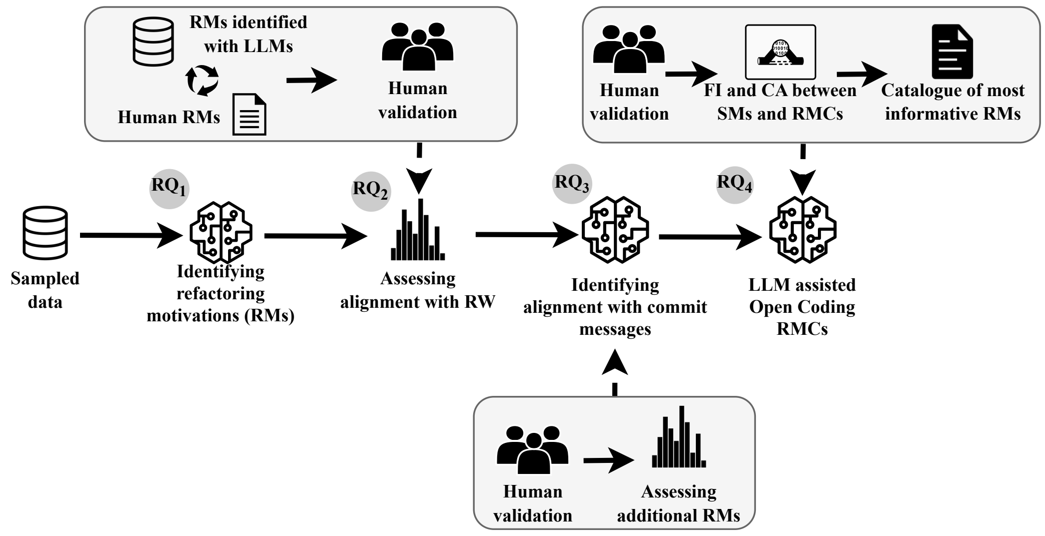

RMs identified with LLMs
Human RMs
Human validation
Human validation
FI and CA between SMs and RMCs
Catalogue of most informative RMs
RQ1
RQ2
RQ3
RQ4
Sampled data
Identifying refactoring motivations (RMs)
Assessing alignment with RW
Identifying alignment with commit messages
LLM assisted Open Coding RMCs
Human validation
Assessing additional RMs

Fig. 4. Data analysis process diagram.

academic specialization in technical debt and refactoring, and long-standing research contributions in software maintenance and code quality assessment. The goal was to assess the quality of the model-generated motivations and identify which models consistently produced reasonable outputs. Our validation followed a three-step protocol:

- One expert independently reviewed the same input provided to the LRM and the three validation models (V1–V3), and manually evaluated the correctness of each model's motivation. For each case, and after making a manual decision over the refactoring case, the expert indicated whether they `agreed` or `disagreed` with the LRM's motivation, noted the majority decision among the validation models, and identified the models they considered correct.
- A second expert repeated the same evaluation independently and documented their level of agreement with the first expert's judgments.
- In cases of disagreement between the first two reviewers, a third expert was brought in to assess the same outputs independently. Final decisions were made through majority voting among the three validators.

Since LLM outputs are non-deterministic, and to assess the reliability of the results, we performed this validation strategy in each of the data analysis stages, which involved LLM outputs (see Figure 4).

Prompting Strategy. To guide LLMs in classifying RMs, we adopted chat-based in-context learning, which has been shown to perform comparably to fine-tuning [25]. Each prompt consisted of two components: a **system message** defining the model's role and expected output, and a **user message** providing the input context.

We employed prompt engineering techniques to guide LLMs in classifying RMs. According to the state-of-the-art, in-context learning through chat-based prompting provides similar or better results than the more computationally expensive fine-tuning process [25]. During the in-context learning phase, the system message established the assistant's role and specified the expected output format, and the user message provided the contextual input.

The user message included details a human developer would typically see: the `RT`, a `description` obtained from RMT (RefactoringMiner), the `commit message`, and the relevant `code diff`. Information on refactoring activity and local changes collected from the version-control history of a project provides the LLMs with localized context on the performed refactorings. However, this might lack more holistic details, such as long-term project goals or further architectural design information. We acknowledge this gap, and acknowledge it as a threat to validity in Section 5.

To improve the attention and consistency of the LLM, we adopted the *Chain of Thought* (CoT) prompting [90], employing zero-shot learning [39, 64]. We systematically refined the employed message prompts during the prompt engineering stage for each of the defined RQs and made the system and user message prompts accessible in our replication package (see Data Availability Statement). Similarly, we expand how we used the CoT prompting strategy, as well as the employed JSON prompt format in Appendix B.2.

We allowed the LLMs the freedom to suggest multiple motivations if they existed by not specifying the number of motivations in the user message. Nevertheless, when the LLM output contained more than one plausible rationale, the human validators examined the proposed alternatives and confirmed whether they represented meaningful additional motivations or simple paraphrases. In cases where multiple distinct motivations could reasonably apply, **the validators selected the primary one by examining the specific refactoring type, the code context, and the developer intent inferred from the surrounding changes**. Therefore, our pipeline ensured that the possibility of multi-motivation refactorings was explicitly considered, even though the final labeling reported one motivation per instance for analytical consistency and comparability. While focusing on the primary motivation facilitates quantitative analysis and alignment with prior literature, we acknowledge this choice to underrepresent secondary or less explicit motivations, and therefore, we further discuss it as a threat to validity (see Section 5).

2.4.2 On the capability of LLMs to extract developers' RM (RQ$_1$). To answer RQ$_1$, we feed the LLMs with each of the 385 sampled refactoring observations (see Section 2.3), including the RT, RMT-generated description, commit message, and code diff from the respective commit as the context variants to provide information within the prompt. Consequently, given the obtained answers from the LLMs, we investigate their capabilities to extract developers' RMs based on the following series of evaluations.

Evaluating the impact of adding different context variants within the prompt on the performance of LLMs for identifying RMs (RQ$_{1.1}$). Before the analysis of RQ$_1$, we performed a prompt engineering process to define the prompt that provided the most reasonable RM results from all the adopted LLMs (see Appendix B). Thereby, we set the same prompt structure to feed the LLMs, and within we provide the RT and description captured by the RMT, the commit message, and the commit diff, which are the *baseline* refactoring context variants (see Table 6). This choice is a best effort, considering the available natural language text information in the repository at the stage of the performed refactoring operation. With the prompt structure fixed, different context variant combinations within the prompt can prove the significance of the variants that better improve the accuracy of the LLMs.

For this, we conduct an ablation experiment, a method used to perform sensitivity analysis and thus evaluate the contribution of each of the components within a system by systematically removing it from the model and then quantifying the impact on the performance of the latter [47]. Thus, we replicate the fixed prompt structure with different ablations of the same user message to the LLMs (see Table 6). Precisely, we feed the employed 4 LLMs with the defined 15 different prompt context variant combinations, summing to a total of 60 replications.

Table 6. Ablation experiment variants for context combinations. (**Ablation 15**: Baseline prompt with the initially considered context variants.)

Ablation	Context variant	Ablation	Context variant
1	Refactoring type	9	Refactoring description, Commit message
2	Refactoring description	10	Commit diff, Commit message
3	Commit diff	11	Refactoring type, Refactoring description, Commit diff
4	Commit message	12	Refactoring type, Refactoring description, Commit message
5	Refactoring type, Refactoring description	13	Refactoring type, Commit diff, Commit message
6	Refactoring type, Commit diff	14	Refactoring description, Commit diff, Commit message
7	Refactoring type, Commit message	**15**	**Refactoring type, Refactoring description, Commit diff, Commit message**
8	Refactoring description, Commit diff		

On the one hand, the LRM solely provides, as an output, a natural text reasoning on the identified RM. On the other hand, the validating LLMs provide their agreement with the LRM reasoning, as a binary output, and a natural description of their assessment. The human validators perform the same LLM activity, i.e., extracting the RM from the developer commit, thus creating the human-curated ground truth (GT). Based on the GT, the human experts reviewed the four LLMs' reasoning and agreement (see Section 2.4.1).

The human expert assessment aims at evaluating the model's accuracy as measured by the usual IR accuracy metrics [28]. It is important to note that since all our experiments are based on only refactoring commits, a motivation, either correct or wrong, will always exist. Therefore, per construction, it is not possible to obtain *negatives*, i.e., nor the refactoring could lack a motivation for refactoring (*true negative*) nor the model can assume no motivation for a refactoring (*false positive*), per construction. Therefore, the only possible outcome to produce the confusion matrix is *True Positives* and *False Negatives.* Thereby, accuracy metrics that rely on false positives and true negatives cannot be computed. Hence, we consider as accuracy metrics commonly used *Recall*, *F1 Score* and *False Negative Rate* (FNR) [28].

Based on the defined ablation experiments and the computed accuracy metrics, we define the following hypotheses to test the impact of the performed sensitivity analysis:

- $H_{0.1}$: *There is no statistically significant difference in the accuracy of the employed LLMs across the employed different context variant combinations.*
- $H_{1.1}$: *There is a statistically significant difference in the accuracy of the employed LLMs across the employed different context variant combinations.*

Consequently, if there was statistically significant evidence on the defined hypotheses, we are interested in testing the pairwise relationship among the different combinations of ablations across different LLMs. Therefore, we derive the following hypotheses:

- $H_{0.2}$: *There is no statistically significant difference in accuracy between prompt variant combination A and prompt variant combination B.*
- $H_{1.2}$: *There is a statistically significant difference in accuracy between prompt variant combination A and prompt variant combination B.*

To perform the statistical inference on the drawn hypotheses, we first examine whether the accuracy scores computed for all the ablation experiments across LLMs follow a normal distribution, and thus decide whether using *parametric* or *non-parametric* tests [27]. Therefore, we draw the following hypotheses:

- $H_{0\mathcal{N}.1}$: *The distribution of the accuracy scores across different ablation experiments is normally distributed.*

- $H_{1\mathcal{N}.1}$: *The distribution of the accuracy scores across different ablation experiments is not normally distributed.*

We test $\boldsymbol{H_{0\mathcal{N}.1}}$ using Anderson-Darling (AD) test [5]. AD tests whether data points are sampled from a specific probability distribution, in this case, a normal distribution. According to Mishra et al., [49], the Shapiro-Wilk (SW) test [76] would be more appropriate when using smaller datasets with fewer than 50 samples. However, our dataset is large enough to use AD, the more powerful statistical test for detecting most departures from normality [82].

If there is statistically significant evidence on the normal distribution of the data, we test H_1 with the parametric ***repeated-measures*** **ANOVA** test [14]. In the opposite scenario, we would perform hypothesis testing with the non-parametric **Wilcoxon signed-rank test** (WT) [91]. Similarly, if H_1 demonstrates significant differences in accuracy scores across ablation experiments, we test H_2 with parametric multiple **paired t-tests** if the data were normally distributed, or with the non-parametric **Dunn's all-pairs test** [22] (DT). The DT test is a post hoc test that compares the differences between all groups within a dataset. Following recent research, we select the critical value of ($\alpha = 0.05$) to assess the significance of the inference results [71].

Evaluating the accuracy of LLMs on extracting developers' RMs (RQ$_{1.2}$). With the ablation experiment, we identify the best performing context variant combination. Subsequently, to assess the LLM's capability to identify developers' RM, we must employ this prompt combination to evaluate their performance. Therefore, we employ the same performance assessment strategy defined for the ablation experiment in the previous RQ, this time implemented with the best-performing prompt. Similarly, we compute the accuracy of the models with *Recall*, *F1 Score*, and *False Negative Rate* (FNR) as previously reported.

On the limitations in LLMs performance when identifying RMs (RQ$_{1.3}$). To answer RQ$_{1.3}$, we qualitatively and quantitatively analyze the LLMs' erroneous outputs across all the sampled observations when they deviated from the GT (human-validated ground truth). We focus on characterizing the nature of the mistakes made by the LLMs and understanding the conditions in which the LLMs failed. For that, we analyze three main dimensions:

- *Disagreement Severity*: We quantify the extent to which LLMs disagreed with the GT based on the total number of models that did not match the RM reported in the GT. Thus, we detect whether a model tends to identify the dimension of the disagreement across the entire analyzed sample of refactoring observations.
- *Information Density*: For each refactoring observation, we compute the information density score based on the log-transformed total number of words included across all the prompt refactoring context variables text data [52]. This enables us to determine whether certain LLMs are more sensitive to sparse or highly condensed information, and whether LLMs' limitations are associated with a lack of contextual misalignment.
- *Error Category*: The authors perform open-coding on the errors performed by the LLMs that did not align with the GT. Thus, we categorize the main reasons or error types reflecting common LLM failure modes.

2.4.3 Alignment and Extension of RMs with the Reference Work (RQ$_2$-RQ$_3$). To build upon RQ$_1$, we now investigate the level of alignment of the RMs identified with the LLMs with respect to the ground-truth motivations identified by Silva et al. [79].

This comparison involved checking for logical consistency, identifying discrepancies, and determining whether the identified RMs extended or complemented the ground truth. It is important to note that while our validated dataset included all RTs mineable by the latest RMT version, the ground truth only covered those listed in Table 7, which presents the frequencies analyzed and originated by the work performed by Silva et al. [79]. Table C.1 provides the names and abbreviations of such RTs.

Therefore, the validation process was limited to RTs analyzed in Silva et al. [79]. Consequently, we pair the sampled RTs and their identified RMs with the ground truth RM counterparts, resulting in a total of 758 RM comparison pairs. From this set, we perform human validation on a statistically significant sample of 198 pairs, with a 95% confidence level and 5% margin of error [73] (see Section 2.4.1).

While the goal of RQ_2 consists of the alignment between the ground-truth RMs and those now identified, we need to validate such results through manual validation, and further assess the reliability of the performed validation. Therefore, and to assess the reliability of the validation, we measure the inter-rater agreement (IRA) between the two raters (human and LLM) using **Cohen's Kappa** (κ) [17], which is well-suited for two-rater scenarios. Unlike Fleiss' Kappa [30], which is designed for multiple raters, Cohen's κ allows us to measure the agreement between a single LLM and a human validator. We interpret the results using Landis and Koch's guidelines [42], where κ values range from less than 0 (no agreement) to 0.81–1 (almost perfect agreement).

Moreover, we conjecture the following hypotheses on the symmetry of the agreement:

- H_{0S}: *The disagreement between raters is evenly distributed.*
- H_{1S}: *The disagreement between raters is not evenly distributed.*

To test such hypotheses, we use **Bowker's test** of internal symmetry [11], a non-parametric statistical test designed to evaluate whether the distribution of disagreements between two raters is symmetric across a square contingency table. In other words, it checks whether the frequency of disagreements in one direction (e.g., human says A, LLM says B) is statistically different from the reverse (LLM says A, human says B). This is particularly useful when dealing with categorical data involving more than two outcome categories, as is the case in our RM classification. Bowker's test extends the well-known McNemar test [46], which is limited to 2x2 tables, by allowing for multi-class comparisons. We select Bowker's test specifically due to the higher dimensionality of our data, which includes multiple motivation types, making it a more suitable and robust choice for assessing symmetry in rater disagreement.

Finally, we are keen to investigate whether LLMs can expand the state-of-the-art by providing new motivations for previously identified ones in the reference study [79] (RQ_3). In this context, we prompt the LLMs with the identified RM, the ground truth RM and developer explanation, as well as the involved commit message, and task them to classify identified RMs as either `related` or `not related` to the ground truth motivations, and if related, if they `expanded` the motivation.

2.4.4 *Software Metrics reflecting Refactoring Motivation (RQ_4).*

This section investigates the degree to which the considered software metrics (SMs) can reflect developers' refactoring motivations (RM). Consequently, we measured the feature importance (FI) and the correlation between SMs (software metrics) and RMs. To such end, and since RMs are notorious for being a human-free text explanation of the aims and insights that lead developers to perform refactoring, we need a categorical, encodable variable. Thus, and in contrast to traditional qualitative analysis, our study employs LLM-assisted open-coding to derive a taxonomy of refactoring motivation categories (RMCs), with posterior human validation.

Therefore, in this RQ, we (1) perform open-coding and define RMCs, and, for the latter, calculate (2) FI and (3) correlation level for the calculated SMs.

Open-Coding. We derive a taxonomy of RMCs from 385 identified RMs. We prompt the LLMs to perform open-coding iteratively, proposing and validating categories derived from the RMs

Table 7. Definitions of the considered refactoring types [31] with unique ground truth motivations (#) [79].

Type	#	Definition
EM	11	Takes a clump of code and turns it into its method.
MovC	9	Move the class to its new folder on the source tree.
MA	2	All attributes matching the selected attribute name on tags with the selected tag name may be moved inward toward a subtag of a given name.
RPack	3	Renames the name of the selected project package.
MM	5	Creates a new method with a similar body in the class that it uses the most. Either turn the old method into a simple delegation or remove it altogether.
IM	3	Puts the method's body into the body of its callers and removes the method.
PUM	1	Moves methods with identical results on subclasses to the superclass.
PUA	1	Moves attributes with identical results on subclasses into the superclass.
ESup	3	Creates a superclass and moves the common features to the superclass.
PDM	1	Given a field only used by some subclasses, it moves the field to those subclasses.
PDA	2	Given an attribute only used by some subclasses, it moves the attribute to those subclasses.
EI	3	Given two classes having part of their interfaces in common, it extracts the subset into an interface.

with the option to introduce new categories, thus simulating the common human practice when performing open-coding. Human validation was performed over a statistically significant sample (95% confidence level, 5% margin of error) afterwards. We provide more details on the conducted open-coding in Appendix B.3.

Feature Importance (FI). To evaluate each the influence of SM (see Table 4) on RMCs, we rank them by their feature importance (FI) [72], a standard feature selection technique used in prior studies to measure the importance of a variable to describe the behaviour of the predicted value after fitting the model [67, 69]. Following prior research [69, 87], we calculate the FI importance of SMs with two commonly known machine learning algorithms, such as Random Forest (RF) [12], and Extreme Gradient Boost (XGB) [15].

The XGB model is an ML algorithm that is part of the family of gradient boost methods [53]. Based on the ensemble learning methodology, i.e., combining the knowledge of multiple models on the same task, the XGB algorithm employs multiple decision trees in a greedy format, where each of the subsequent trees corrects or refines the wrong results from the previous tree. Through this technique, the algorithm defines weights for the included independent variables that require a higher emphasis [16]. Therefore, it allows for analyzing the explainability of its model training by quantifying the feature importance of the fitted model variables, in our case, the collected software metrics.

The RF model also belongs to the ensemble learning family and is built based on the ensemble of decision trees [13]. Based on random sampling, the RF algorithm performs multiple decision trees, and the output is the aggregated value from the results obtained in the trees performed [12]. Based on the scores from each of the parallel employed trees, it enables the representation of feature scores to quantify the level at which each of the model features used to fit the trees contributes to improving the performance of the model.

For RF, we use two established FI measures: Mean Decrease in Accuracy (MDA) and Mean Decrease in Gini (MDG) [33, 54]. MDA reflects how much the model's accuracy drops when a

Table 8. Spearman's ρ and Kendall's τ interpretation.

	Perfect	Strong			Moderate			Weak			Zero
ρ	±1.0	±0.9	±0.8	±0.7	±0.6	±0.5	±0.4	±0.3	±0.2	±0.1	0
τ	±1.0	±0.8	±0.7	±0.6	±0.5	±0.4	±0.3	±0.2	±0.1	±0.05	0

specific variable is permuted—higher values suggest higher importance. MDG, on the other hand, measures how much a variable helps reduce node impurity in decision trees—again, higher scores imply greater relevance. For XGB, we rely on Information Gain (IG) as the FI measurement, which quantifies how much each variable contributes to the model relative to the total contribution of all variables [15]. A higher IG indicates a stronger impact.

Since feature dimensionality can influence the learning process and FI scores, we evaluate three scenarios: one using only process metrics, one using only product metrics, and one combining all SMs.

Correlation Analysis. To investigate the statistical relationship between RMCs and SMs, we conduct a correlation analysis. As part of testing our RQ$_4$, we first assess the distribution of the RMC data to determine the most appropriate correlation coefficient [28]. To test the distribution data, we define the following hypotheses:

- $H_{0\mathcal{N}.2}$. *The detected RMCs are normally distributed.*
- $H_{1\mathcal{N}.2}$. *The detected RMCs are not normally distributed.*

We follow the rationale explained in $RQ_{1.2}$ and adopt the AD test to test the normality on $H_{0\mathcal{N}.1}$. According to the test results, we could reject the null hypothesis in all cases. Therefore, to test $H_{0\mathcal{C}.1,0\mathcal{C}.2}$, we must rely on a non-parametric test. More specifically, we select Spearman's ρ coefficient [81], which evaluates the monotonic relationship between two variables, and tests if one variable's change leads to the other variable's change, in the same direction (positive correlation) as well as in the opposite direction (negative correlation); and Kendall's τ coefficient [92], which evaluates the similarity between two variables through their ordinal association. The correlation coefficient denotes a positive correlation when the observations for the pair of variables present a similar rank, and conversely, it presents a negative correlation if the orderings are dissimilar. To interpret ρ and τ, we adopt Dancey and Reidy interpretation [19] (see Table 8).

Finally, since we rely on many hypothesis tests, we mitigate the risk of Type I error or *family-wise* error at the level of α [34]. When multiple tests are conducted simultaneously, the probability of falling into the aforementioned error increases. Therefore, we adopt the Bonferroni correction test [93], addressing the risk by adjusting globally the significance level ($\alpha' = \frac{\alpha}{m} = \frac{0.05}{574} = 0.00008711$), where α is the standard pre-defined significance level, i.e., usually equal to 0.05, and m is the number of performed hypothesis operations. Nevertheless, existing literature acknowledges that Bonferroni is overly conservative when dealing with large-scale hypothesis testing, therefore we also adjust the p-value with the Benjamini-Hochberg (BH) method [9] to control the False Discovery Rate (FDR). Unlike more conservative methods like Bonferroni correction that control the Family-Wise Error Rate (FWER), the BH method allows for more discoveries, i.e., rejections of null hypotheses, by limiting the expected proportion of false positives among those discoveries. It works by ranking p-values and comparing them to an increasing threshold based on their rank, offering a balance between discovery and error control [18].

3 RESULTS

This section presents the findings of our investigations and addresses the RQs:

3.1 Evaluating the capability of LLMs to extract developers' RMs (RQ_1)

To answer RQ_1, we analyzed the set of 385 refactoring observations drawn from the sampling stage performed during the data collection. In the following paragraphs, we present our results on the evaluation of LLMs' capabilities to extract developers' RMs.

3.1.1 Evaluating the impact of adding different context variants within the prompt on the performance of LLMs for identifying RMs ($RQ_{1.1}$). With the setting for the ablation experiments defined (see Section 2.4.2), we fed the LLM pipeline with 385 refactoring observations based on 15 different ablations of our prompt. The collection of ablations consisted of every possible combination of the refactoring context variants considered in our study, including the baseline ablation, which contained all the refactoring content variants initially considered to feed the LLMs (see Table 6). First, we calculated the accuracy of the LLMs with respect to the human GT, and thus we obtained the results for the Recall, the F1-Score, and the FNR accuracy metrics (see Table 10). Before conducting the designed statistical inference, we explored the distribution of incorrect answers provided by each LLM across all the experimented ablations. Table 11 reports the mean, median, and standard deviation of the number of refactoring observations in which the LLMs failed to provide a correct RM with respect to the GT across the analyzed ablations. Overall, all the models presented a mean value of more than 50 refactoring observations in which they did not provide correct refactoring motivations, with V1 (Mistral NeMo) presenting the highest number of errors on average (70.86). While V3 (Phi-4) presented the best results with the minimum mean error value, it was the LRM (Marco-o1) that presented the lowest median in terms of erroneous RMs across different ablations.

Subsequently, to assess whether the results were normally distributed, and thus define whether performing parametric or non-parametric statistical inference, we tested $H_{0\mathcal{N}.1}$ with the AD test and we could reject it for the distribution of the three metrics (A^2 = 3.2271, p-value < 0.0001 for Recall, A^2 = 4.7701, p-value < 0.0001 for F1-Score, and A^2 = 3.2271, p-value < 0.0001 for FNR). Hence, to assess the statistical significance of the impact of different refactoring context variants on the employed LLMs, we tested $H_{0.1}$ with the WT test for each of the accuracy metrics, respectively, and we could reject the null hypothesis for each of the metric distributions with a p-value of less than 0.0001, and a χ^2 of 49.4786. Therefore, we could claim that **there is a statistically significant difference in the accuracy of the employed LLMs across the different context variant combinations**.

Building upon this, and to assess the difference in terms of accuracy between multiple pairs of ablation experiment variants, we tested $H_{0.2}$ with the non-parametric DT test, and we could reject it in 5 out of 105 tested pairs for each of the analyzed accuracy metrics. Given the dimensionality of the performed statistical testing, while we provide all the results within the shared replication package, Table 9 presents the results for the statistically significant pairwise results. Based on the resulting statistically significant pairwise tests, we could reject the null hypothesis in less than the 5% of the pairwise combinations for each of the metrics.

Interestingly, on the one hand, **the baseline ablation did not result in statistically significant improvements to the LLMs in terms of accuracy metrics in any of the tested multiple pair comparisons.** On the other hand, **none of the ablations yielding statistically significant results resulted in reporting significant improvements to the LLMs in terms of accuracy metrics with respect to the baseline ablation**.

Additionally, we revisited the raw results from the ablation experiments and calculated the share of refactoring observations in which none of the LLMs provided the right results with respect to the GT (Ground Truth). Interestingly, on average across all ablation experiments, none of the LLMs provided correct answers with respect to the GT in 23.04% of the refactoring observations. Furthermore, it was **the baseline ablation experiment the one that presented the minimum**

Table 9. All pairs Dunn's test statistically significant results ($p < 0.05$).

Abl. A	Abl. B	Metric	Z	p-value
5	4	Recall	3.8470	0.0126
12	4	Recall	3.6243	0.0304
11	4	Recall	3.5939	0.0342
5	3	Recall	3.5838	0.0356
4	2	Recall	-3.5231	0.0448
5	4	F1	3.8470	0.0126
12	4	F1	3.6243	0.0304
11	4	F1	3.5939	0.0342
5	3	F1	3.5838	0.0356
4	2	F1	-3.5231	0.0448
4	2	FNR	3.5231	0.0448
12	4	FNR	-3.6243	0.0304
11	4	FNR	-3.5939	0.0342
5	3	FNR	-3.5838	0.0356
5	4	FNR	-3.8470	0.0126

Table 10. Summary statistics of Recall (R), F1-score, and FNR across all ablations.

	Median			Mean			Std. Dev.		
Abl.	R	F1	FNR	R	F1	FNR	R	F1	FNR
1	0.5987	0.7489	0.4013	0.6006	0.7504	0.3994	0.0196	0.0153	0.0196
2	0.7416	0.8516	0.2584	0.7435	0.8527	0.2565	0.0240	0.0158	0.0240
3	0.3351	0.4975	0.6649	0.3312	0.4897	0.6688	0.1107	0.1262	0.1107
4	0.2416	0.3891	0.7584	0.2468	0.3956	0.7532	0.0166	0.0211	0.0166
5	0.7584	0.8626	0.2416	0.7610	0.8643	0.2390	0.0132	0.0085	0.0132
6	0.6312	0.7739	0.3688	0.6383	0.7788	0.3617	0.0345	0.0254	0.0345
7	0.6091	0.7569	0.3909	0.5974	0.7471	0.4026	0.0476	0.0379	0.0476
8	0.6623	0.7968	0.3377	0.6578	0.7933	0.3422	0.0299	0.0219	0.0299
9	0.7610	0.8641	0.2390	0.7318	0.8430	0.2682	0.0852	0.0591	0.0852
10	0.3701	0.5395	0.6299	0.3701	0.5389	0.6299	0.0491	0.0524	0.0491
11	0.7494	0.8567	0.2506	0.7494	0.8566	0.2506	0.0168	0.0110	0.0168
12	0.7623	0.8648	0.2377	0.7461	0.8534	0.2539	0.0631	0.0422	0.0631
13	0.6260	0.7696	0.3740	0.6227	0.7669	0.3773	0.0405	0.0308	0.0405
14	0.6429	0.7821	0.3571	0.6474	0.7852	0.3526	0.0485	0.0357	0.0485
15	0.6987	0.8226	0.3013	0.7143	0.8327	0.2857	0.0468	0.0312	0.0468

Table 11. Summary statistics on the occurrences of mistaken refactoring motivations per model.

Model	Mean	Median	Std. Dev.
Marco o1 (LRM)	55.1333	47	35.4700
Mistral NeMo (V1)	70.8667	65	32.3217
DeepSeek R1 (V2)	67.8667	75	31.9707
Phi 4 (V3)	50.9333	52	24.0905

share of cases with incorrect answers from the LLMs (8.83%), while **the ablation considering which only considered the commit message as the context variant (see Table 6) presented the highest share of observations with no LLM providing right answers (69.35%)**.

1. Which context variant to choose?

Only **5% of the tested multiple pair comparisons between ablations** reported statistically significant results. While some ablations reported significant improvements, **there were no improvements with respect to the baseline ablation**.

3.1.2 Evaluating the accuracy of LLMs on extracting developers' RMs ($RQ_{1.2}$). Given the results of $RQ_{1.1}$, none of the defined ablations involved improvements in the LLMs or outperformed the efficiency of the baseline ablation. This reflected a lack of empirical evidence supporting the superiority of any reduced-context ablation over the originally designed prompt configuration. Therefore, we proceeded using the baseline ablation as the default prompt variant for the remainder of the study.

Thereby, to answer our $RQ_{1.2}$ we now evaluate how accurately each LLM identified RMs under the baseline ablation. Figure 5 reports the accuracy metrics obtained by each of the four LLMs when employing the baseline ablation.

Marco-o1, our LRM model, achieved the highest performance across metrics, with a Recall and FNR of 0.78 and an F1-score of 0.88, indicating stronger capability to correctly identify developers'

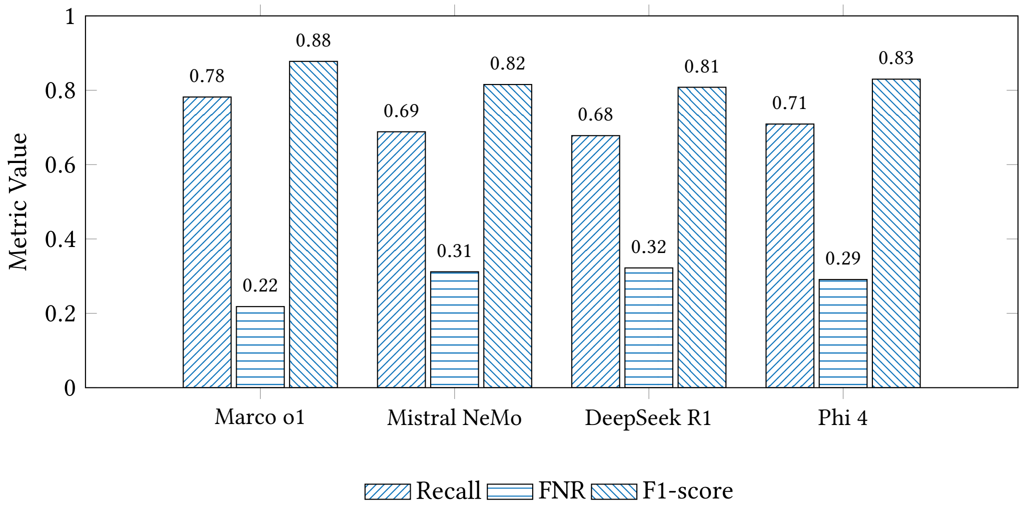


Fig. 5. Performance of LLMs under the baseline prompt configuration, comparing Recall, FNR, and F1-score across models.

RMs. The V3 model, *Phi-4*, yielded balanced results presenting the second-best scores with a Recall and FNR of 0.71 and an F1-score of 0.83. *Mistral NeMo* and *DeepSeek R1* models did not show a significant lower performance as their results achieved a Recall and FNR values of 0.69 and 0.68, and F1-scores around 0.82 and 0.81, respectively.

2. Which LLM performed the best?

Marco-o1, as the model adopted as LRM, provided the best results with a Recall and FNR of 0.78 and an F1-score of 0.88.

3.1.3 On the limitations in LLMs performance when identifying RMs (RQ$_{1.3}$). To answer our RQ$_{1.3}$ we performed a data-driven interpretation of the insights resulting from the human manual validation of the analyzed sample. First, when investigating the disagreement severity among LLMs, we observed that **over half of the sample refactoring observations presented full agreement with the human GT (50.10%)**. Conversely, **20.52% of the refactoring observations presented severity level 1**, which translated into cases where one single LLM was disagreeing with the GT (Ground Truth), this was closely followed by the 16.90% of the cases presenting **a systematic disagreement of three LLMs disagreeing with the GT**. Further cases in which all the LLMs disagreed with the GT, or two of them disagreed, represented 0.003% and 11.43% of the analyzed refactoring observations. These results suggest that, while many LLM errors appear to be isolated as only one LLM out of four disagrees with the GT, multiple cases in which at least 3 LLMs disagreed with the GT represent **shared blind spots across models**. Notably, this last aspect reveals potential **common limitations in LLM reasoning and contextual understanding**.

Regarding the results obtained when calculating the Information Density in each of the refactoring observations, we grouped the results by the disagreement severity level observed. Thus, we aimed at exploring the naive association between the information density of the prompts provided to the

LLMs and their disagreement severity. On average, prompts with the highest information density score (6.76) resulted in full disagreement with the GT from all the LLMs. While simultaneous disagreement severity levels of 3, 2, and 1 LLMs described a descending curve pattern with log-scaled results of 5.84, 6.04, and 5.83, respectively, cases in which no LLM disagreed with the GT showed a mean information density level of 5.92, allowing for more dense prompts than those in which some of the LLMs disagreed. Nevertheless, overall, these results suggest that **higher information density within the context provided to the LLMs is naively associated with a higher disagreement severity**.

Lastly, we observed the resulting error categories and their number of occurrences from the analyzed refactoring observations. Overall, from the 189 refactoring observations in which at least one LLM disagreed with the human GT, the open-coded results revealed that **most LLM failures arise from substantive reasoning mistakes rather than superficial misunderstandings**. For instance, the category *Factual Error* (43.39%) demonstrated that LLMs are limited in accurately interpreting the context provided in the prompt, as they provided answers that were simply incorrect. The second largest category, *Context misalignment* (20.10%), in which LLMs represented tendencies toward over-elaborated or misplaced reasoning. Further errors, *Concept Deviation* (17.99%), which demonstrated the difficulty LLMs face in distinguishing between semantically similar motivations. Other less represented errors, such as *Unreal verbosity* (10%) and *Vagueness* (4.76%), represented tendencies toward over-elaborated or not well specified reasoning. Finally, a small share of errors resulted in the *Limited Scope* category (3.70%), highlighting that LLMs might force the answer with a valid, yet limited conclusion, thus demonstrating a lack of understanding of the topic by the LLM.

3. Identified LLM limitations

The observed disagreement severity patterns indicate **shared blind spots across LLMs**. This appears to be affected by the information density, as **higher contextual density naively associates with increased disagreement severity**. The open-coded error categories demonstrate that **most failures stem from context interpretation issues**.

3.2 Refactoring Motivations: Alignment With Previous Studies (RQ$_2$)

To answer RQ$_2$, we analyzed a total of **758** pairs of RMs (Refactoring Motivations), where each pair consisted of two RMs, one from the RMs identified in our study and the other one corresponding to the RM reported by Silva et al. [79], both related to the same RT (Refactoring Type). Our findings focus on the alignment with the reference study [79] and the agreement with human experts, followed by an examination of the underlying RMs behind the observed disagreements. Therefore, we needed to assess the alignments based on RM pairs resulting from the same RT.

Considering the volume of retrieved RMs, applying a 95% confidence level and a 5% error rate, we sampled a subset of **198** pairs: **136** generated by the LRM (Large Reasoning Model), while **62** generated by V3 (see Section 2.4.3). Interestingly, V3 was found to produce more complete and reliable RMs in these fallback cases. As a result, the final composition of our *corpus* of RMs was composed by **69%** LRM-generated and **31%** V3-generated.

To assess the degree of alignment between these LLM-generated RMs and the ones available in the reference study, we tasked the LLMs and three human experts to evaluate such alignment. In 59 cases out of 198, both the humans and the LLM judged no alignment with the reference study ("No-No", True Negatives), while 97 out of 198 showed agreement with both judging alignment ("Yes-Yes", True Positives). However, there were 42 cases of disagreement: in 8 instances, humans

Table 12. Representativeness level for the refactoring types sampled in the manual validation for RQ_1.

RT	Representativeness	# Agreement	# Disagreement
EM	27.77%	30	25
MM	17.17%	6	28
MovC	13.64%	12	15
MA	8.59%	8	9
IM	8.08%	5	11
PUM	5.05%	8	1
PDA	5.05%	6	4
ESup	3.54%	4	2
PDM	3.03%	4	2
RPack	3.03%	1	5
PUA	2.53%	5	0
EI	2.53%	5	0

(#: Refers to frequency)

Table 13. LLM Motivation Alignment Agreement Contingency Table.

LLM	Human	Frequency	%
No	No	59	29.79
No	Yes	8	4.04
Yes	No	34	17.17
Yes	Yes	97	48.99

Table 14. Human Validated LLM RM Alignment with reference study RMs.

Classification	Frequency	%
Extends	44	22.22
No	49	24.74
Yes	105	53.03

identified alignment that the LLM did not ("Yes-No", False Positives), whereas, in 34 instances, the LLM assumed alignment where humans did not ("No-Yes", False Positives) (see Table 13).

The LLMs generated RMs aligned well with human judgment in most cases, showing agreement in 156 out of 198 cases (~80%), considering the combination of the true positives and true negatives (see Table 13). Furthermore, we measured IRA (Inter-Rater Agreement) via Cohen's kappa between LLM and humans, resulting in a moderate agreement ($\kappa = 0.567$) and it was statistically significant ($p < 0.0001$) (see Table 15). Moreover, according to Bowker's symmetry test, the **disagreement was systematic** ($\chi^2 = 16.10$, $p < 0.0001$), suggesting the LLM consistently identifies diverse RMs concerning the human ones in terms of interpretation or overlooking. Therefore, we can affirm that there exists a statistically significant, moderate agreement between LLM and human RMs, but systematic differences in interpretation remain.

We were also keen to investigate whether specific refactorings were more challenging to motivate and agree upon. The most frequently represented operations were EM (Extract Method) with a 27.77%, MM (Move Method) with a 17.17%, and MC (Move Class) with a 13.64% (see Table 12). Less common types included RPack (Rename Package) with a 3.03%, PUA (Pull Up Attribute) with a 2.53%, and EI (Extract Interface) with a 2.53%.

PUM (Pull Up Method) showed the strongest agreement between LLM and human RMs (8 out of 9 pairs aligned), followed by PUA (Pull Up Attribute) and EI, both with perfect agreement (5 out of 5). In contrast, MM presented the most disagreement, with 28 disagreements out of 34 comparisons. IM (Inline Method) (11 out of 16) and MC (15 out of 27) also showed significant disagreement. These results suggest that LLMs struggle more with refactorings involving method or class movement.

Based on human annotations, **97 out of 198 RMs (approximately 49%) aligned with those reported in the reference study**.

Finally, to understand the rationale between humans and LLMs, we manually analyzed the 42 disagreement cases, categorizing them into **five recurring disagreement categories**, each reflecting a fundamental difference in how LLMs and humans approach the interpretation of RMs (see Table 16). Overall , LLMs tend to reason based on localized code context, e.g., method names or syntax-level cues, likely because they lack access to full project information. In contrast, human

Table 15. Agreement Test Human Motivation Vs LLM

Agreement Test	**Kappa**	**Std Err**	**Lower 95%**	**Upper 95%**
	0,567	0,057	0,455	0,679
Asymptotic Test	**Prob > z**	**Prob> $\|Z\|$**		
	$<, 0001$	$<, 0001$		
Bowker's Test: Symmetry of Disagreement	**ChiSquare**	**Prob>ChiSq**		
	16,095	$<, 0001$		

Table 16. Summary of manually detected disagreement categories between LLM extracted motivations and those from the reference study.

Disagreement category	**Description**	**Occurrences**
Different Focus of Refactoring Granularity	Human focuses on attributes, classes, or packages while LLM focuses more localized structures such as methods.	201
Intent Misalignment: Structural vs Functional	Human focuses on structural aspects (e.g. clarity, visibility...) while LLM emphasizes on a more functional perspective (e.g. testability...).	118
Semantic vs Syntactic Understanding	Human centres on semantic changes to clarify intentional code statements (e.g. ownership, package naming...) while LLM refers to class organisation or simple syntactical clarity.	46
Future vs Present Orientation	Humans often refer to future needs in their motivations (e.g. extension, scalability), while LLM focuses on immediate operations such as current testing needs.	26
Interpretation of Refactoring Scope	Human views refactoring as modular reorganization while LLM sees the operation as an isolated change.	10

raters apply a more holistic perspective, considering architectural structures and long-term design goals when assessing RMs.

For instance, in the Future vs Present Orientation category, humans often justify changes based on anticipated future needs, such as extensibility or scalability, whereas LLMs focus on immediate benefits, like current testability or readability. Similarly, in the Different Focus of Refactoring Granularity category, LLMs interpret changes at the method level, while humans consider higher-level structures, such as classes or packages. These patterns suggest that LLMs are proficient in identifying surface-level improvements, but they frequently miss deeper, strategic rationales that require an understanding of broader software design intent.

Table 17. Summary of LLM Motivations that Extend Human-Reported Motivations.

Characteristics of LLM Extensions	Example Description	Occurrences
Broader Scope	LLM identifies additional aspects of refactoring (e.g., moving attributes in addition to methods).	40
Maintainability and Readability Emphasis	LLM highlights code maintainability and readability, extending beyond human focus on code functionality.	27
Structural Clarity	LLM explicitly addresses improvements to the structural clarity and dependency injection of the code.	7
Enhanced Detail and Precision	The LLM motivation clarifies human-provided motivations with more precise and detailed reasoning.	3
Explicit Testing and Flexibility Context	LLM explicitly includes motivations related to improving testing and code flexibility.	3
Comprehensive Renaming Context	Human motivation focuses narrowly on renaming; LLM elaborates underlying reasons for renaming.	2

4. Aligned or not Aligned?

Only **47% of the LLM-generated RMs matched the prior reference study** according to human evaluation. While there was an **80% agreement between LLMs and humans overall**, systematic differences emerged; LLMs rely heavily on **local code cues**, while humans factor in **architectural and strategic goals**. Certain refactorings, like **Pull Up Method**, saw strong alignment, but **Move Method** and **Move Class** revealed persistent LLM struggles, emphasizing the need for deeper context understanding to align fully with past studies.

3.3 Refactoring Motivations: Extensions of Previous Studies (RQ$_3$)

To investigate how LLM-generated motivation content compares with prior studies, we examined the degree to which they extend the state of the art. **105 out of 198 analyzed cases (53%)** showed direct matches with RMs previously reported in the reference study (see Table 14). Moreover, in **44 cases out of 198 (22%)**, the LLMs also **agreed** in the motivation reported in the reference study, but they also inferred **meaningful extensions or refinements** with information such as context-specific or fine-grained justifications.

Conversely, **49 cases (25%)** did **not align with any existing motivation**, hinting at **potentially novel rationales**.

More specifically, we identified seven distinct rationales where LLM-generated RMs expanded human-reported ones (see Table 17), thus introducing **new motivational elements by LLMs that were not reported by human-reported motivations regarding the same refactoring observation analyzed**. These include adding missing technical details, e.g., clarifying the benefits of a specific refactoring operation that human RMs overlooked, expanding the scope of changes, e.g., covering both method and attribute moves, and emphasizing aspects such as readability, structural clarity, and testability.

5. LLMs extend the state of the art

While **53%** of the RMs matched known reasons from past studies, the remaining **47%** revealed either **extended** (22%) or **entirely new** (25%) RMs. These included overlooked technical details, broader refactoring scopes, and added focus on **readability, structure, and testability**, highlighting that developers may be driven by **richer and more diverse RMs** than previously collected.

Moreover, and to enrich our qualitative assessment, we investigated the extent to which developers explicitly acknowledge refactoring in commit messages. For that, we performed a regular expression matching for the keyword family "refactor*" on all commits with RefactoringMiner-detected refactorings (see Appendix G). We acknowledge this approach to be the best effort, as it might not capture the totality of the self-admitted refactoring commits given the existing different practices in which developers acknowledge the performance of refactoring activity within their software development processes, and therefore, discuss its implications in Section 5 as a threat to validity. Based on the performed text mining, only 4% of commits in the full dataset and 12% in the manually validated sample were classified as self-admitted refactorings. This confirms that developer motivations are rarely explicitly documented, given our mined data, reinforcing the need for approaches such as LLMs, capable of performing code context deduction.

3.4 Software Metrics reflecting Refactoring Motivations (RQ$_4$)

This section investigates the relationship between developer RMs for refactoring and SMs, to understand whether SMs can capture aspects of the same underlying developer rationale. Our aim is threefold: (1) to categorize the RMs extracted from LLMs via open-coding, (2) to rank the SMs based on their ability to capture human rationale among RMCs, and (3) to analyze the statistical correlations between the identified RMCs and the computed metrics.

3.4.1 Open-Coding Refactoring Motivations into Categories. We first conducted LLM-assisted open-coding on 385 sampled refactoring observations to identify recurring RMs, and validated the results through human validation. This process yielded **167 distinct RMs**, i.e., unique motivations among the total of 385 RMs identified, which we grouped into **14 RMCs** (see Table 18) for the sake of encoding the existing collection of distinct RMs, and thus render FI and correlation analysis more explainable. These categories encompass common refactoring rationales such as **Code Clarity and Readability (CCR)**, **Code Simplification and Redundancy Reduction (CSRR)**, and more specialized goals like **Encapsulation and Abstraction**. Moreover, based on performing a miscellaneous definition of the presented RMCs, Table 18 also includes the classification that the authors performed of the refactoring motivations identified by Silva et al. [79] with respect to each RMC, thus allowing us to map the extent to which previous state-of-the-art refactoring motivations align with findings resulting from this study.

Figure 6 summarizes the frequency distribution of the extracted RMs. We present the frequency of total identified RMs grouped per RMC (blue), as well as the frequency of distinct or unique RMs, thus showing not only how many motivations were identified per RMC but also how many different motivations each RMC encapsulates. Among all, **CCR** (30.91%) and **CSRR** (24.04%) were the most frequently occurring RMCs. However, when considering only the unique instances, CSRR (20.36%) slightly surpassed CCR (17.36%), indicating a richer diversity of simplification-related RMs.

Among the most dominant RMCs, **Code Clarity and Readability (CCR)** emerged as the most frequent, with 119 occurrences. This category encapsulates RMs centred on enhancing the comprehensibility, abstraction, and ease of reading code. It includes cases where developers aim to refactor by renaming variables for clarity, extracting methods to reduce cognitive load, or organizing code to align with human-readable logic.

Following closely, **Code Simplification and Redundancy Reduction (CSRR)** appeared 81 times and reflects RMs that emphasize minimizing unnecessary complexity. This includes reducing duplicated logic, collapsing verbose constructs, or eliminating parameters and variables that no longer serve a purpose. Together, CCR and CSRR account for over half of the observed RMs, reinforcing that LLM-generated RMs often prioritize the legibility and minimalism of code.

The third most common category, **Maintainability and Modularity (MM)** (35 occurrences), reflects efforts to improve the long-term evolvability and structural organization of the codebase. RMs in this category often involve modularizing components, encapsulating change-prone logic, and enhancing the separation of concerns to support sustainable maintenance.

Beyond these primary themes, several RMCs address more specific technical aspects. For example, **Encapsulation and Abstraction (EA)** (24) focuses on isolating responsibilities and reducing the surface of external interactions, which includes practices like hiding implementation details or reducing class coupling. Similarly, **Testing and Reliability (TR)** (19) relates to improving the testability of code, often through simplifying logic or making control flows more deterministic.

A notable portion of RMs also fell into **Other Specialized Goals (OSG)** (20), which act as a catch-all for domain-specific or niche objectives that do not neatly fit into other categories. This includes specialized algorithmic refactoring, compliance with domain-specific constraints, or integration-related concerns. Less common but still essential RMCs included:

- **Security and Safety (SS)** (15), addressing concerns like thread safety, null safety, or eliminating dangerous constructs.
- **Exception and Error Handling (EEH)** (13), which involves improving how code deals with failures or unexpected states.
- **Type and Parameter Handling (TPH)** (13), focusing on method signatures, type safety, and semantic correctness.
- **Support for (New) Functionalities (SF)** (12), where the motivation is driven by expanding or improving system capabilities.
- **Structural Reorganization (SR)** (11), representing architectural restructuring actions like moving classes or extracting responsibilities.
- **Consistency and Standardization (CS)** (10), which targets uniformity in naming, formatting, or style to align with project standards.
- **Performance and Resource Management (PRM)** (8), dealing with optimizations such as memory usage, threading, or execution time.
- **Design Principles and Patterns (DPP)** (5), the least represented, where the motivation stems from aligning code with well-known design patterns or architectural principles.

Such a taxonomy of RMs lays the foundation for subsequent analysis, allowing us to study how measurable SMs correlate with or indicate the presence of specific RMCs.

3.4.2 Investigating Metrics Importance for Refactoring Motivations Categories. Understanding what drives developers to initiate refactoring activities is crucial for building intelligent, context-aware recommendation systems, for instance. In this final stage to answer RQ_4, we investigate which SMs, both process and product metrics, are most indicative of specific RMCs. We employ two state-of-the-art machine learning models, Random Forest (RF) and Extreme Gradient Boosting (XGB), to compute feature importance scores and isolate the metrics that most strongly influence model predictions, and two statistical tests, Spearman and Kendall, to measure the correlation of such metrics with the RMCs.

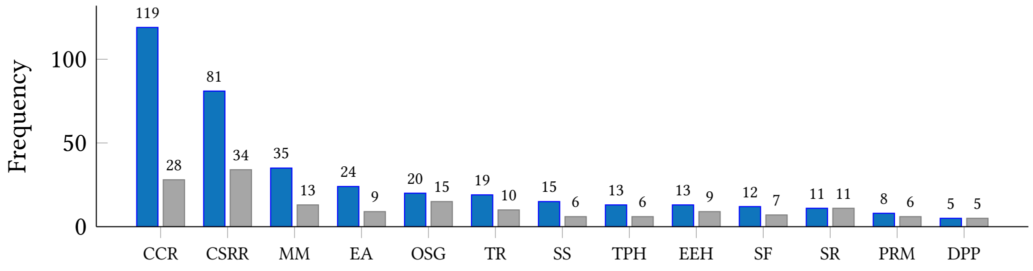


Fig. 6. Frequencies for extracted total motivations (**blue**) and unique motivation instances (**grey**).

Table 19. Refactoring Types Frequencies (#) for CCR RMC

RT	**#**	**RT**	**#**	**RT**	**#**	**RT**	**#**
EV	8	EAMM	6	RM	5	EM	4
RC	4	RV	4	CRT	3	RParam	3
MovC	3	ACM	3	RAWL	3	AMA	3
RLWP	3	IM	2	MA	2	CPT	2
RCA	2	MAIM	2	RPack	2	RPWL	2
RemP	2	MARM	2	MARC	2	SM	2
IV	2	CVT	2	RVM	2	MPack	2
RMA	2	MAA	2	RA	2	AVM	2
RPM	2	ACA	1	RenP	1	MARA	1
RCWT	1	SA	1	TWR	1	MovM	1
RA	1	MerA	1	ESub	1	AAM	1
MCon	1	RGWD	1	RAA	1	RVA	1
RPA	1	MCode	1	SClass	1	CAT	1
PDA	1	LP	1	RCM	1	RAM	1

(**RT**: Refactoring Type)

Table 20. Refactoring Types Frequencies (#) for CSRR RMC

RT	**#**	**RT**	**#**	**RT**	**#**	**RT**	**#**
EV	7	RemP	7	IV	5	MParam	3
MV	3	PUM	3	RPA	3	RVM	3
MAIM	2	MerC	2	MerM	2	RCA	2
RMA	2	RTET	2	RVA	2	RLWP	2
SC	2	CAT	1	CCAM	1	CPT	1
CRT	1	CTDK	1	ExA	1	EM	1
ESup	1	IA	1	IM	1	IC	1
LP	1	Mcat	1	MCon	1	MCA	1
MVA	1	PV	1	PUA	1	RAA	1
RCM	1	RPM	1	RAWV	1	RCWT	1
RGWD	1	RVWA	1	SP	1		

(**RT**: Refactoring Type)

RF Results. We analyzed **process metrics** importance scores according to RF in terms of Mean Decrease in Accuracy (MDA) and Mean Decrease in Gini (MDG). MDA and MDG measure how much each metric contributes to improving the model's predictive performance (see Section 2.4.4).

From the **MDA perspective**, which reflects how much the model's accuracy decreases when a feature is randomly permuted, the most informative process metrics were **ADEV** (the cumulative number of active developers per file) with a score of 7.37, and **MINOR** (number of minor contributors to a file) with 6.70, indicating that diversity and activity of contributors are particularly relevant signals when trying to predict the target outcome in the model (see Figure 7).

Conversely, **MDG**, which reflects how each variable contributes to the purity of the decision tree splits (i.e., how well it separates the data), top-ranked **EXP** (average developer experience in the project, 38.37), **OEXP** (ownership experience on a file, 38.28), **COMM** (number of commits per file, 36.35), and **NSCTR** (developer spread across packages, 35.94). Hence, we note that long-term developer familiarity with the project and the intensity of change activity in files and packages are key indicators used by the model to capture structural patterns in the evolution of the codebase that tend to co-occur with certain motivational categories.

Table 18. Description of the open-coded refactoring motivation categories (**blue**: Occurrences in our study, grey: Occurrences from the reference study [79]).

Motivation Category	**Description**	**Occurrences**
Code Clarity and Readability (CCR)	Motivations aiming to improve the readability, abstraction, and understandability of the code.	119 22
Code Simplification and Redundancy Reduction (CSRR)	Focus on reducing complexity, eliminating duplication, and streamlining code.	81 1
Maintainability and Modularity (MM)	Focuses on long-term maintainability and modular decomposition of software components.	35 3
Encapsulation and Abstraction (EA)	Deals with isolating responsibilities and minimizing external dependencies or access.	24 0
Other Specialized Goals (OSG)	Motivations serving niche, technical, or domain-specific purposes.	20 115
Testing and Reliability (TR)	Refactorings aimed at improving code testability and reliability.	19 6
Security and Safety (SS)	Motivations ensuring safer, more secure code, such as null safety or thread safety.	15 0
Exception and Error Handling (EEH)	Improving how exceptions and errors are managed in the codebase.	13 0
Type and Parameter Handling (TPH)	Type safety, parameter handling, and semantic correctness of method inputs.	13 25
Support (New) Functionalities (SF)	Enhancing functionality and introducing new features.	12 0
Structural Reorganization (SR)	Movement or reclassification of structural elements.	11 92
Consistency and Standardization (CS)	Aligning code with standards or maintaining consistent patterns.	10 3
Performance and Resource Management (PRM)	Efficiency, memory, and threading improvements.	8 0
Design Principles and Patterns (DPP)	Use of design patterns and separation of concerns.	5 0

In summary, these findings reveal that both **developer-related characteristics** (e.g., experience and ownership) and **collaboration dynamics** (e.g., number and type of contributors) play a central role in shaping the model's understanding of the data.

Regarding the importance of **product metrics**, from the **MDA** perspective, the top-ranking product metrics were **NS** (number of modified subsystems, 5.37) and **NUC** (number of file changes up to the current commit, 4.25). Such metrics reflect structural and evolutionary dimensions of the system, i.e., **NS** reflects the scope of change across architectural units (i.e., subsystems), while **NUC** emphasizes how frequently a file is subject to modification (see Figure 8). Their high importance suggests that the **extent of system modularity** and **change-proneness of code artifacts** are critical indicators for identifying relevant behavioral patterns such as RMC (Refactoring- or Maintenance-related Changes).

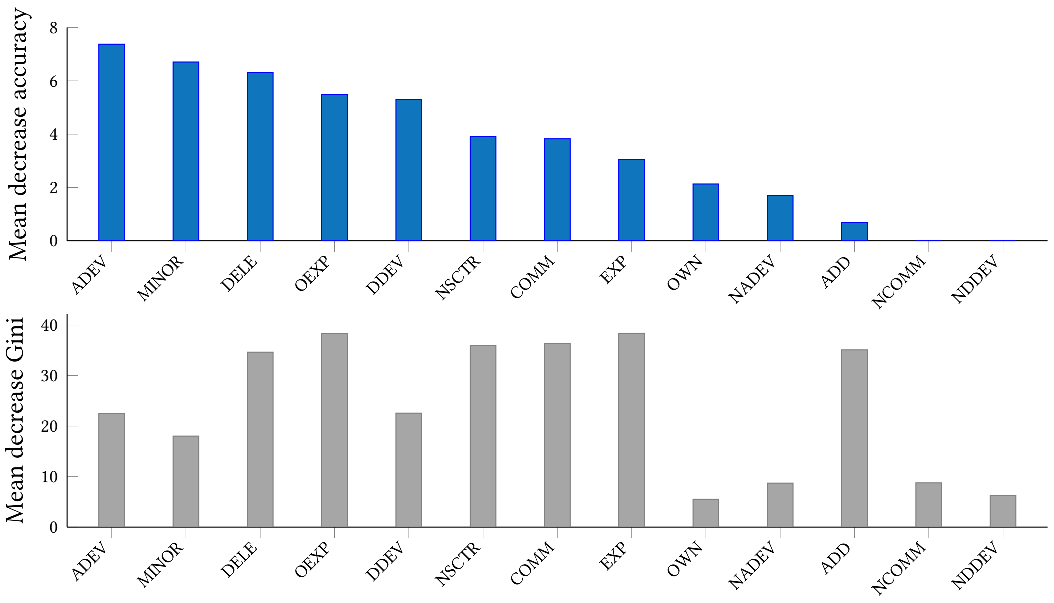


Fig. 7. MDA and MDG importance results for Random Forest model trained with process metrics.

On the other hand, the **MDG** scores point to a different set of influential factors. The most prominent metric was **COMREAD (Val)** (18.69), which assesses the **comprehensive readability** of a class by incorporating textual, structural, and visual cues. This was followed by **NS** (17.84), **AGE** (average time since the last change, 17.12), **LT** (lines of code before the change, 17.01), and **LD** (lines deleted, 16.73). These findings underline the role of **code readability, code age, and size-related factors** in shaping the model's predictions. More specifically, the high relevance of **readability** suggests that the ease of understanding code, possibly influencing developer decisions, could be a key factor in detecting RMCs. Overall, the differences between MDA and MDG emphasize that while **structural evolution and change intensity** dominate in MDA-based importance, **code quality and maintainability dimensions** such as readability and code churn gain prominence when viewed through the lens of MDG.

Finally, regarding the **combined importance of both product and process metrics**, **MDA results** emphasize the influence of developer-related characteristics, with **SEXP** (number of commits a given developer performs in the considered package containing the given file) and **NS** emerging as the most influential metrics (scores of 5.65 and 5.47, respectively). These are followed closely by **MINOR** (number of contributors who contributed less than 5% of a given file up to the considered commit), **NDEV** (number of distinct developers who changed any of the files involved in the commits where the given file has been modified), and **ADEV**, further underscoring the significance of both developers' **experience** and **collaboration dynamics** in predicting refactoring, or main developers-related change categories (RMCs) (see Figure 9). This blend of human and structural factors suggests that both the *quantity* and *diversity* of developer contributions shape the likelihood of code evolution events. On the other hand, the **MDG scores** highlight another important layer of insight. The top metrics, **OEXP** (percentage of code lines authored by a given developer in the project) with 13.84, **EXP** (13.53), and **COMREAD** (12.49), collectively represent **developer ownership**, **project experience**, and **code comprehensibility**. These were followed closely by **NS** and **NSCTR** (both 11.97), showing that the structural breadth of change across subsystems

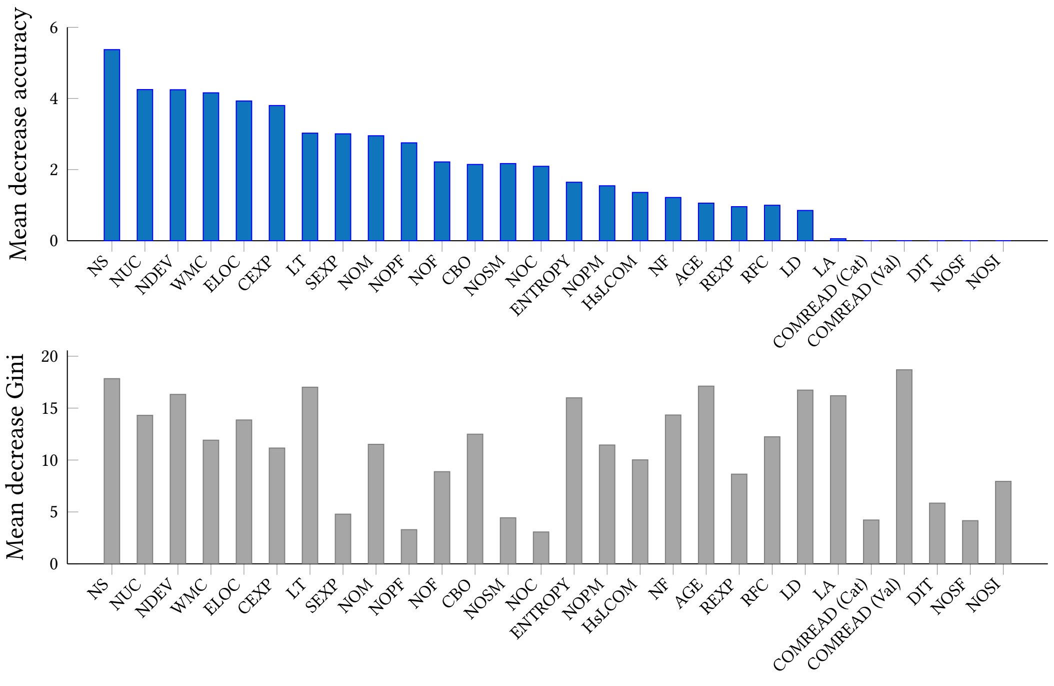


Fig. 8. MDA and MDG importance results for Random Forest model trained with product metrics.

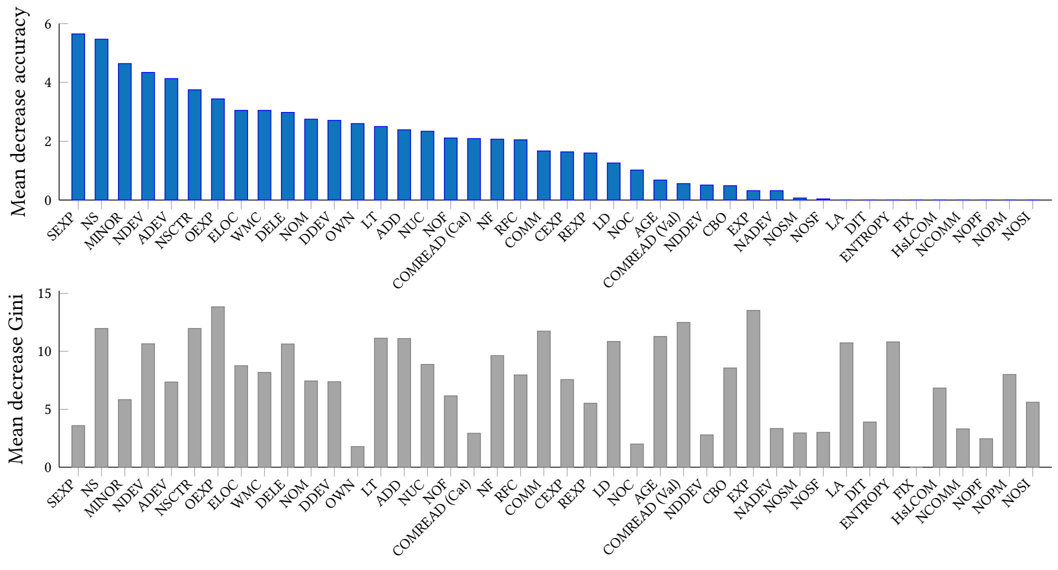


Fig. 9. MDA and MDG importance results for Random Forest model trained with process and product metrics.

and packages also plays a key role. Altogether, the MDG-based ranking complements the MDA findings, reinforcing the conclusion that RMCs are influenced by a **combination of code quality, developer engagement, and structural complexity**.

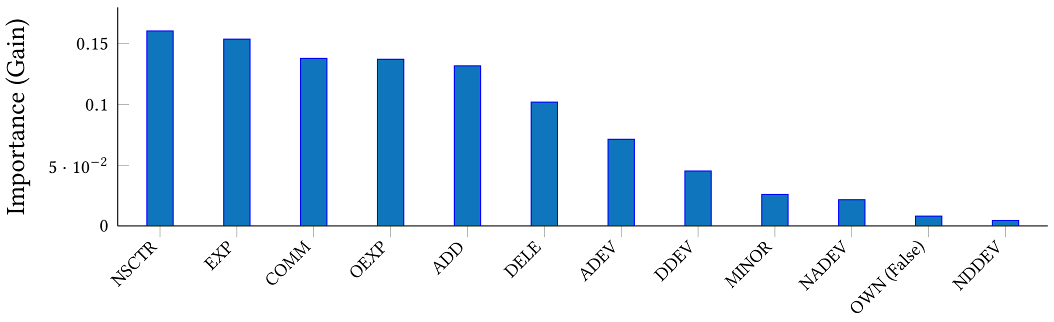


Fig. 10. Bar chart for process metrics importance from the trained XGB model.

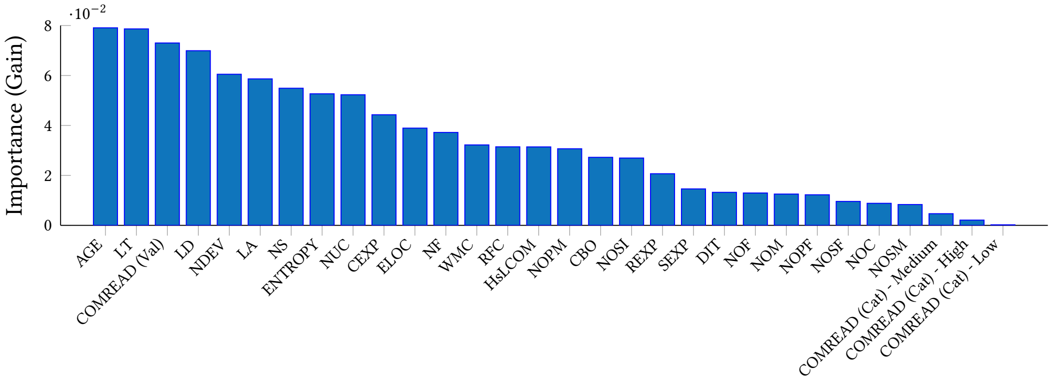


Fig. 11. Bar chart for product metrics importance from the trained XGB model.

XGB Results. XGB measures the importance of metrics in terms of Information Gain (IG). Regarding **process metrics**, top-ranked **NSCTR** (16.05%) and **EXP** (15.35%), with **COMM**, **OEXP**, and a second instance of **ADD** scoring between ~13–14% (see Figure 10). These results are consistent with the RF-based findings, reaffirming that **developer experience** and the **breadth of file/package-level changes** are dominant predictors of RMCs. The repetition of OEXP highlights its consistent impact across learning models. Overall, these results strengthen the hypothesis that **developer history and cross-cutting modifications** are key explanatory factors in process-driven code evolution.

Regarding **product metrics**, **AGE** of a file (7.90%) and its **length before modification (LT, 7.86%)** emerged as the most informative (see Figure 11). These were followed by **COMREAD (Val)** (7.30%), **LD** (number of lines removed from the given file in the considered commit) (6.99%), and **NDEV** (6.05%). These findings mirror the earlier MDG-based RF results, where code **maturity**, **comprehensibility**, and **developer interaction** were pivotal. The emphasis on **readability** and **historical change patterns** indicates that certain design and evolution characteristics make a file more prone to refactorings or maintenance.

Finally, when combining all metrics in the XGB training (see Figure 12), **COMREAD** surfaced as the most important variable (6.45%), affirming the **centrality of code comprehensibility** in predicting RMs. This was followed by **NSCTR** (6.25%), the metric capturing change spread across packages, and **COMM** (5.52%), **EXP** (5.18%), and **OEXP** (5.13%). The consistency across modeling techniques and metric types provides strong evidence that **code readability, developer**

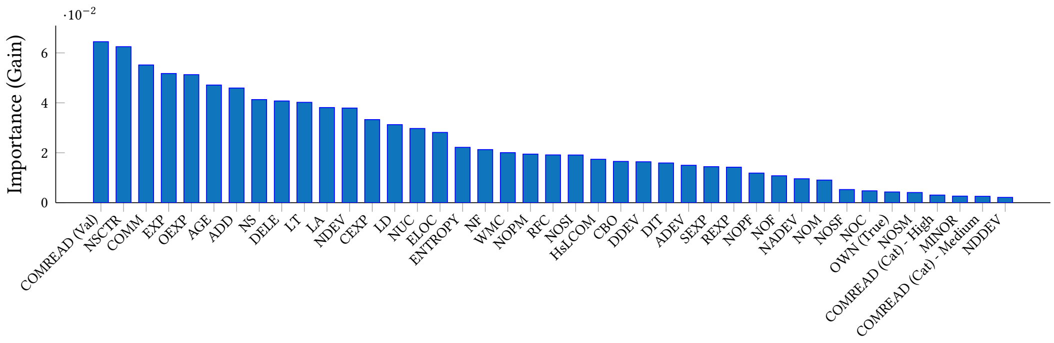


Fig. 12. Bar chart for product and process metrics importance according to XGBoost.

experience, and change scope are jointly the most critical signals for understanding why RMCs occur.

6. Metrics Importance

Developer experience, code readability, and change scope are the strongest predictors of RMCs, with Random Forest emphasizing ownership and commits, and XGB highlighting readability and change propagation.

3.4.3 Investigating Metrics Correlation with Refactoring Motivation Categories. The AD (Anderson-Darling) test results allowed us to reject the null hypothesis of normal data distribution ($H_{0\mathcal{N}.2}$), therefore we must rely in non-parametric correlation analysis using **Spearman's** ρ and **Kendall's** τ.

Spearman Correlation Results. Figure 13 displays the correlation matrix, both uncorrected and adjusted with Bonferroni and BH (Benjamini-Hochberg) corrections. Out of 574 tests, we could reject the null hypothesis H_0 in 39 uncorrected correlations. After correction, we could only reject the null hypothesis H_0 in **17 cases** following the BH method.

Kendall Correlation Results. The uncorrected Kendall τ allowed us to reject the null hypothesis H_0 in 39 out of 574 cases (see Figure 14), consistent with Spearman's. However, no significant results remained after applying Bonferroni or BH corrections.

Metrics-based Results. The correlation analysis revealed several noteworthy, albeit weak, trends between specific metrics and RMs. Metrics like **COMREAD**, **NF** (number of modified files), **NSCTR**, and **ENTROPY** showed weak positive correlations with **code clarity (CCR)** motivations, suggesting that developers may prioritize readability and the modular spread of changes when aiming to clarify code. Interestingly, **COMREAD** and **NOF** (number of fields declared in a class) were negatively correlated with **code standardization (CS)** motivations, possibly indicating that overly complex or large field declarations may hinder standardization efforts.

For **code simplification and removal of redundancy (CSRR)**, weak positive associations were found with **DELE**, **LD**, **NADEV**, and **SEXP**, reflecting a tendency to simplify code where there is significant churn and developer interaction. **LA** positively correlated with **abstraction-related RMs**, while **EXP** showed a weak negative relationship, possibly indicating that less experienced teams may abstract more aggressively.

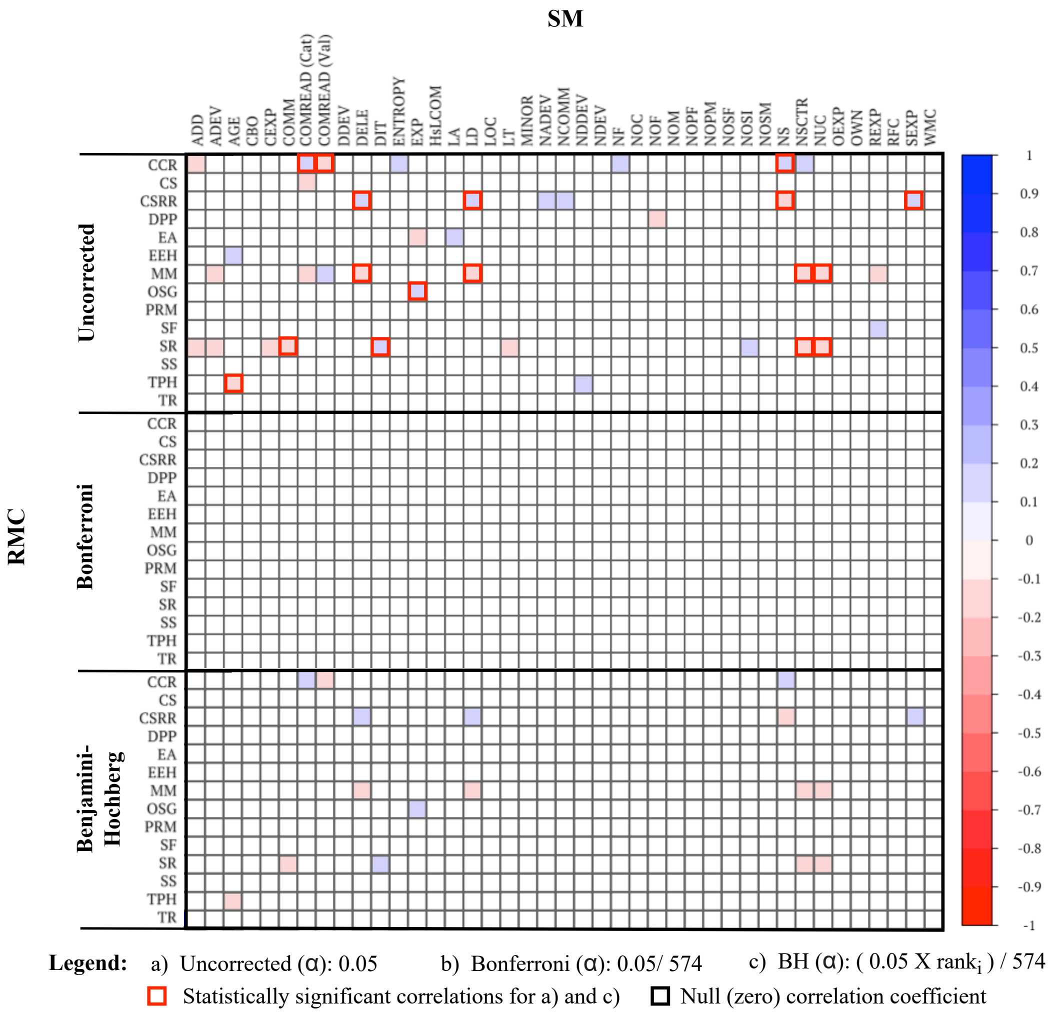


Fig. 13. Heat-map representing the Spearman correlation matrix of RMCs with SMs for the uncorrected, Bonferroni corrected, and BH corrected probabilities.

The **AGE** metric was negatively linked to **technical performance and hygiene (TPH)** motivations, suggesting that older files might be neglected in performance-focused refactoring. For **structural reorganization (SR)**, **DIT** showed a weak positive correlation, highlighting deeper inheritance trees as a refactoring trigger, whereas **NUC** and **NSCTR** showed negative ones, implying that files with broader or frequent changes may be less targeted for structural reshaping.

Lastly, **REXP** correlated weakly positively with RMs tied to **supporting new functionalities (SF)**, while **COMREAD**, **ADEV**, and **REXP** showed weak negative correlations with **maintenance and modernization (MM)**, suggesting that highly readable or frequently changed code may not always be the primary target for modernization.

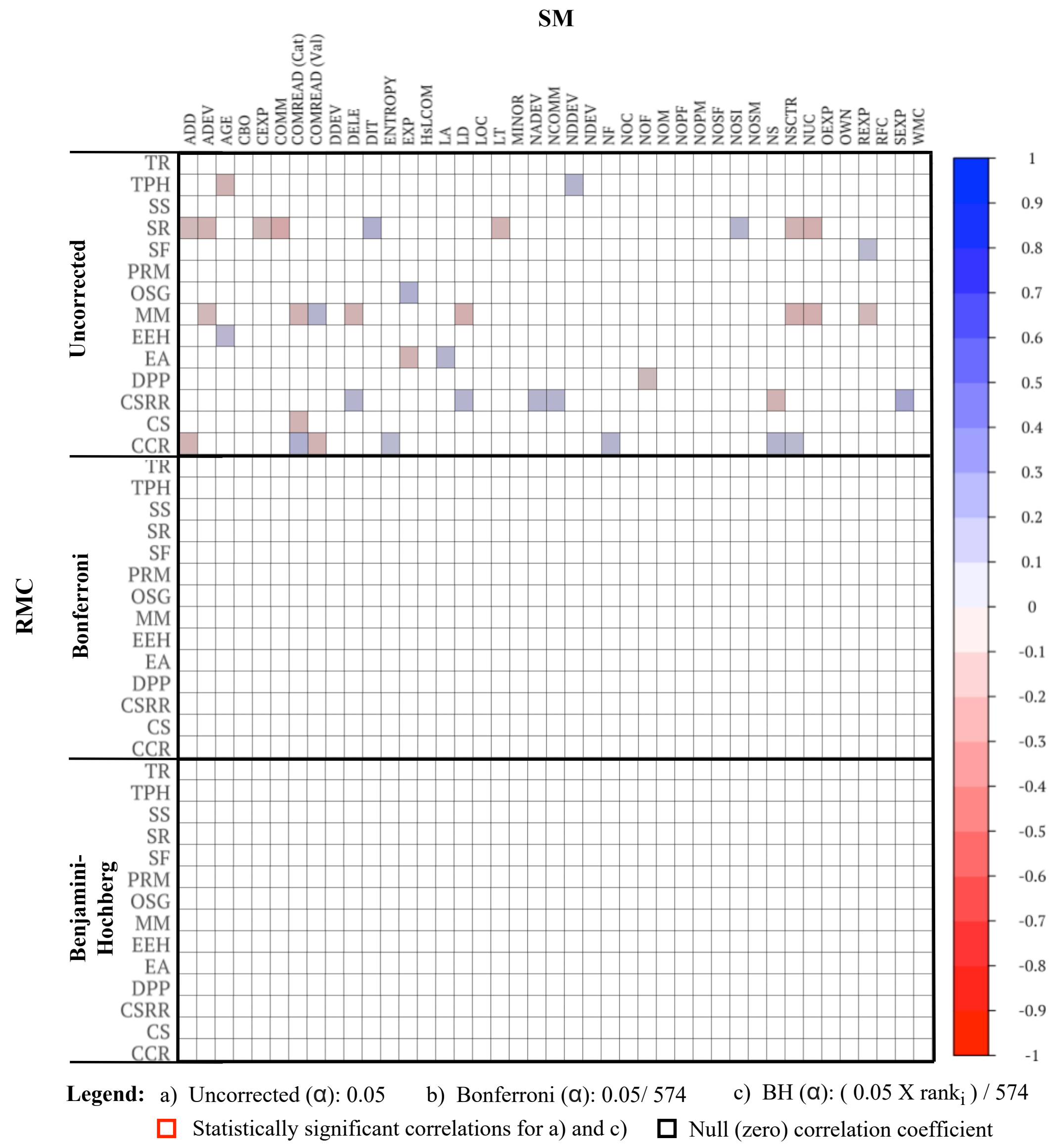


Fig. 14. Heat-map representing the Kendall correlation matrix of RMCs with SMs for the uncorrected probabilities.

7. Scarce Correlation and Statistical Significance

Out of 574 tests, Spearman's ρ identified 17 significant correlations after BH correction, revealing weak but interpretable links, e.g., code clarity (CCR) with readability and entropy, and maintainability (MM) with deletion and change frequency. Conversely, Kendall's τ yielded no significant results post-correction.

Answering RQ$_4$. Our findings revealed that both process and product metrics play complementary roles in predicting RMs. RF's MDA favored contributor dynamics and modularity, highlighting metrics like ADEV, MINOR, and NS, while its MDG emphasized experience (EXP, OEXP) and code change intensity (COMM, NSCTR). Product-wise, structural complexity (NS, NUC) and readability (COMREAD.Val) emerged as important signals. XGB corroborated these trends: for process metrics, developer experience, and file/package change spread (EXP, NSCTR) stood out; for product metrics, file age (AGE), size (LT), and readability again topped the list. Notably, across both models, developer engagement, code comprehensibility, and structural change scope consistently ranked as key predictors of RMCs. The findings underscore a dual influence of social and technical dimensions in shaping code evolution. Similarly, metrics such as EXP, OEXP, and SEXP do not approximate personal intention but rather capture structural patterns in how code evolves. For example, high SEXP—indicating that a contributor has long-standing involvement in a subsystem—was frequently associated with simplification-oriented refactorings, as experienced developers possess the contextual understanding needed to reduce accumulated complexity. Conversely, lower ownership (OEXP) combined with a higher proportion of minor contributors (MINOR) tended to align with readability-driven refactorings, where developers improve comprehensibility before adding new functionality. Metrics like ADEV and NDEV, reflecting the number of developers interacting with a file, often corresponded to motivations targeting maintainability and coordination, such as naming updates or structural clarifications in collaboratively edited areas. Overall, these interpretations demonstrate that the metrics identified as important by RF, XGB and the conducted correlation analysis represent meaningful contextual signals of developer refactoring motivation, capturing both human and technical dimensions of code evolution and helping to explain why specific refactoring motivation categories tend to arise in particular development contexts.

8. Metrics and Refactoring Motivations

Both RF and XGB models converge on three main predictors of RMs: **developer experience**, **code readability**, and **change scope**. RF highlights developer ownership and activity (e.g., EXP, OEXP, COMM), while XGB reinforces the role of code comprehensibility (COMREAD) and structural evolution (NSCTR, AGE). These findings collectively underscore the intertwined effect of human factors and code characteristics on driving refactoring decisions.

4 DISCUSSION

We leverage the following section to discuss the key findings of our study by relating the outcomes to every research question and identifying actionable implications for researchers and practitioners. We also address the applicability of our results with respect to future refactoring recommendation systems.

Our results for RQ$_1$ revealed that LLM performance in identifying developers' RMs strongly depends on the completeness of the provided context. In fact, the conducted ablation experiment (RQ$_{1.1}$) demonstrated that removing context variants does not significantly improve the LLMs' accuracy, thereby indicating that, at least, **there is no evidence indicating that LLMs benefit from more concise and synthesized prompt context**, and neither in the opposite direction. Based on this premise, we followed our analysis with the baseline prompt context, and analyzing the accuracy of the adopted LLMs (RQ$_{1.2}$), ***Marco o1* achieved the strongest performance**, yet there were no significant differences among the remaining LLMs, confirming that **although results look**

promising, LLMs remain inaccurate as developer RM identifiers. Moving into RQ$_{1.3}$, evidence on nearly half of the disagreements involving one or two models failing simultaneously denoted **shared blind spots across LLMs**. These failures were associated with a higher information density received by the LLMs in the prompt, suggesting LLMs' difficulty in understanding long and dense contexts. When open-coding the potential error categories identified among the mistaken LLMs' answers, results showed that **misinterpretations and conceptual confusions dominate, rather than superficial misunderstandings**. Overall, results in RQ$_1$ demonstrated that while the LLMs' capabilities for RM identification are promising in many cases, **they still hold systematic weaknesses such as context overload or limitations for deeper reasoning**.

1. **Main insights from RQ$_1$: Full-context prompts remain essential**, as experimented ablations did not provide accuracy improvements. **LLM accuracy is promising**, but RM identification cannot yet fully depend on LLM deduction capabilities. Identified limitation from LLMs while identifying RM entangle **shared reasoning blind spots** as multiple LLMs failed together, higher information density demonstrated naive association with higher disagreement severity. In addition, most mistakes stemmed from **factual inaccuracies**.

In addition, since motivations may differ depending on whether refactoring is performed independently or as a part of a broader process or task, we additionally mined our refactoring dataset to estimate the share of bug-fixing refactoring commits (see Appendix F). Using regular expression-based pattern matching, we found that out of all refactoring commits, 54% corresponded to issue-fixing activity, while in the manually validated sample, this proportion dropped to 28%, based on the total size of the sample. These results suggest that floss refactorings represent the major share of refactoring activity to the extent of the mined software projects.

2. **Root-Canal vs. Floss Refactoring.**: Refactoring operations opportunistically performed while addressing bugs represent a substantial share of the mined refactoring activity.

As shown in RQ$_2$, LLMs agreed with human judgments in 80% of the cases, but only 47% matched RMs from the reference study. Systematic disagreements emerged, as confirmed by Bowker's test, with LLMs tending to interpret RMs based on local code cues, names, syntax, and immediate readability, rather than broader architectural goals. For instance, RMs for MM and MC saw substantial misalignment, with LLMs overlooking structural intentions behind these changes. In contrast, Pull Up operations had a near-perfect alignment, likely due to their localized semantics.

Human raters, instead, often based their evaluations on project-wide consistency, long-term maintainability, and design abstraction, factors that are invisible in the local code snapshot available to the LLM. This gap reveals that while LLMs can mimic reasoning for simple refactorings, they fall short in capturing RMs rooted in architecture or foresight.

3. **Local vs Structural Reasoning**: LLM-generated RMs should be interpreted as partial approximations. They are effective for routine, localized changes but unreliable for strategic or architectural reasoning. To close this gap, future tools could expand LLM context with project-wide metadata or history-aware embeddings.

According to RQ_3, only 53% of LLM-generated RMs aligned exactly with the prior reference study [79]. However, in 22% of the cases, LLMs enriched the rationale by adding context-specific nuances or detailing effects on testability, naming conventions, or structural clarity. These elaborations were never contradictory; instead, they made tacit developer knowledge explicit.

This ability to extend RMs identified by Silva et al. [79] aligns with the finding that LLMs consistently emphasized clarity, maintainability, and readability. For example, when humans described motivation as "renaming," the LLM often contextualized it in terms of onboarding ease or cognitive load, surfacing deeper reasoning that might otherwise be lost.

4. **Enriching Motivation Clarity**: LLMs offer value by articulating implicit reasoning. Their output can help with onboarding, documentation, or code review, surfacing hidden RMs that developers rarely write down. This supports knowledge retention beyond the act of coding.

The RQ_4 analysis of 385 samples revealed that over 55% of all RMs were aimed at improving code clarity (CCR) or simplifying redundant logic (CSRR). Most RMs stemmed from pragmatic needs, naming clarity, method simplification, and parameter cleanup, rather than enforcing design principles or structural overhauls. Refactoring was thus largely incremental, not architectural.

This aligns with our earlier findings: LLMs were effective when refactorings were localized and incremental, and struggled when RMs depended on broader system-level reasoning. These trends match developer behavior observed in earlier work [23].

5. **Pragmatism Over Principles**: Developers often refactor for short-term clarity, not ideal long-term architecture. Tools that surface refactoring suggestions should prioritize simplicity, understandability, and friction reduction over theoretical design goals.

Our machine learning and correlation analysis inferences revealed that while some metrics (e.g., `EXP`, `OEXP`, `COMREAD`) were top-ranked by importance models (RF/XGB), their correlation with motivation categories was weak and often statistically insignificant. For example, only 17 correlations passed significance after correction in Spearman's analysis, and none did in Kendall's.

This suggests metrics alone lack the semantic richness to explain why developers refactor, which slightly deviates from prior works [60]. However, their role as signals is non-negligible: metrics related to developer engagement, file age, and code readability consistently ranked high in feature importance scores, indicating they can guide or contextualize predictions.

6. **Metrics in a Supporting Role**: Metrics do not reflect motivation directly but can inform LLMs of context and likelihood. They are better used as input features or filters in hybrid systems rather than as standalone predictors.

Our research stemmed from the idea of supporting developers with a catalogue of SMs, informative on the behaviour of the drawn RMCs. A question that naturally arises from this research is whether our findings support the development of LLM-based refactoring recommendation systems. And the response is tentatively affirmative. LLMs showed a satisfactory ability to classify and justify RMs, and they were particularly resilient when justifying simple, localized refactoring intentions. They struggled with architectural or cross-cutting RMs and required human judgment to resolve open cases. The weak correlation between metrics and RMs means that a metric-only system would be unreliable. The only plausible direction is, therefore, hybrid systems: metric-informed LLMs that blend behavioral hints with natural language reasoning to suggest refactorings and explain why they do so.

7. **LLM-Guided Refactoring Systems**: LLMs can be the basis of explanation- and clarity-focused refactoring recommendation systems. To make them more reliable, these systems can include SMs as well as project-level context, so that recommendations reflect both code attributes and probable developer intention.

Our results highlight **key limitations and opportunities for research** in Generative AI-assisted software engineering. First, the systematic differences between LLM and human RMs suggest the need to explore methods for enriching LLM inputs with higher-order architectural or historical context. Researchers should investigate techniques such as long-context prompting, integration of architectural models, and retrieval-augmented generation (RAG) that can provide LLMs with broader system-level signals. The observed ability of LLMs to extend developer RMs also opens avenues for studying how to formalize tacit knowledge and how to balance verbosity with precision in generated justifications. Additionally, the weak correlation between metrics and motivation categories underscores a gap in our current abstraction tools. Research should re-evaluate which metrics meaningfully reflect cognitive developer behavior or develop new hybrid indicators that better represent intent. Finally, the potential of hybrid LLM-metric models invites empirical validation: how can we measure trust, usability, and impact of such systems in real-world settings?

Practitioners can leverage LLMs as assistants in tasks that require expressing, documenting, or reviewing code change rationales. Our results show that LLMs are especially useful in surfacing developer intent for routine, local refactorings and in articulating implicit reasoning that is often left undocumented. Teams can integrate LLMs into code review pipelines or commit templates to clarify RMs and make them traceable over time. However, caution is needed: for refactorings that touch architectural layers or involve non-local effects, LLMs should be treated as suggestive rather than authoritative. Moreover, while SMs remain useful, especially those reflecting experience and activity, they should complement, not substitute, LLM output. A practitioner-facing system that combines

LLM insights with contextual metrics can support better onboarding, reduce misunderstanding in reviews, and help preserve architectural consistency by making rationale explicit and shareable.

5 THREATS TO VALIDITY

This section discusses the threats to validity, including internal validity, external validity, construct validity, and reliability. Moreover, we explain the different adopted tactics [92].

Construct Validity. We acknowledge that using LLMs for *semi-supervised multi-class* classification may pose a threat to the results of this study, as it assumes the AI-based feedback is correct. However, to minimize this threat, we considered using state-of-the-art as well as well-valued distilled LLMs such as Marco-o1 and Mistral NeMo, among others. Similarly, we provided the models with localized version control data related to the RM to support the prompt with project context, which might result in local reasoning output from the LLMs. We considered supporting the models with such a level of local context as the first approach for leveraging LLMs in RM identification, as no previous study had used them for such a task. Silva et al [79] had already considered the threat of using RMT for mining the commits in which the refactorings were performed, as some classifications may be missed. However, we still consider using RMT to provide consistency to our results, as one of our goals is to validate the ground-truth motivations provided by developers. Furthermore, RMT remains the state-of-the-art refactoring mining tool to date. We also identify a threat to the validity of our results in the use of the refactoring commits, and their corresponding RMs, sourced from the reference study, which were derived from projects we did not mine with RMT in our study due to computational resource limitations. We tackled this threat by mining the SM related to refactoring commits in which the affected RMs from the reference study were identified, and hence, we had all the required data to still consider these refactoring observations as ground-truth.

Internal Validity. One of the main outcomes of this study is the publication of a set of refactoring motivations related to code refactoring types already studied by prior work, as well as refactoring types still not investigated. These motivations are based on the combination of LLMs' natural language understanding capabilities and the human validation performed in our study. To better align the results from our study, we had to focus our analysis on a primary motivation per refactoring operation, therefore underrepresenting secondary motivations. Missing motivations from developers and potential motivations yet to be discovered by analyzing further projects may contribute to the possible lack of accuracy. However, we understand that the motivations provided by human developers [79] provide the closest feedback to the real motivations for committing refactoring, and therefore, diminish this type of threat. We acknowledge the existing threat to validity in considering OSS LLMs for the motivation identification stage of the analysis. Leveraging the latest state-of-the-art models depends on the budget and the specific required computational power. To tackle this threat, we searched for open-source alternatives that claim to present similar results as the current state-of-the-art reasoning models. Similarly, future work should explore extended analyses of AI-based refactoring-motivation identification, ideally incorporating multiple motivation scenarios in a more comprehensive qualitative setting. Another concern related to our inclusion of refactorings occurring outside pull requests (PRs), which extends prior reference studies [60]. While this choice expands the scope of mined data, it may raise the question of whether PR and non-PR refactorings differ in ways that could affect the interpretation of our results. With the results from the mined data representing a minority of refactoring operations being related to PR workflows, we consider the sample to remain representative of the overall refactoring landscape. Nonetheless, future work could further explore whether refactoring motivations differ between PR-related and non-PR-related development workflows. Lastly, we also acknowledge the existing threat to validity in considering *regular expression* matching within the commit messages for the issue-fixing commit detection, as well as for the self-admitted refactoring detection. Hence, we acknowledge the use of this technique as a best effort given its complexity, and it is currently an ongoing research topic in Software Engineering research [4, 28].

External Validity. This study only considers open source projects, based on the Java programming language and hosted in GitHub. Therefore, the results of this study, even though they aim to cover a wide range of projects, cannot claim to apply to systems outside the open-source community or projects developed in a different programming language. However, the presented study plan aimed to analyze all the existing refactoring activities of 124 projects and all their mineable refactoring types supported by the adopted mining tool, which, if the results provide a clear guideline of metrics, would provide one of the most consistent recommendation guidelines for refactoring activity in the field.

Reliability. The planned data analysis is presented in a format that aims to provide answers to both the refactoring cases in which the motivation is identifiable and when discrepancies may exist, and therefore, already considered motivations may not fit the refactoring case. Similarly, the choice of statistical tests detailed in this article aims at covering the possible statistical assumptions that differently distributed data may require for performing statistical testing. The source data provides a considerable set of GitHub-hosted projects. Yet, given the existing number of projects published on the mentioned platform, only a small portion of the samples was considered. Therefore, we consider as a threat the hypothetical differences in results if the same analysis were implemented in a different sample of projects. However, the validation of the developer-based motivations through the results of this study would minimize the presented threat, as we understand that the motivations presented by developers are not the result of their performance in the analyzed projects but the result of their entire development career, and hence the product of a larger number of projects.

6 RELATED WORK

This section reviews related work on refactoring recommendations, outlining the current state of research and presenting a detailed comparison with our approach in Table 21.

Software refactoring can improve different aspects of software quality, such as readability and maintainability [55]. However, refactoring is also a time-consuming task and, as such, should be appropriately prioritized, and the scenario concerning refactoring operations should be thoroughly studied [40]. Therefore, to meet practitioners' needs, we must understand their motivations when performing such refactoring operations [79] to better identify the scenarios in which code refactoring is needed.

Wang [89] investigated human motivations for performing refactoring operations by conducting multiple case studies across four software development organizations and with a total of ten interviews with professional software developers. Their work emphasized empirically and qualitatively understanding the reasons why developers choose to refactor code. Through grounded-theory analysis of interview data, their work identified personal motivations such as responsibility for code quality, self-esteem, and unconscious habits, as well as external motivators such as perceived procedural value, reduction of assigned tasks, and threats of punishment at work. Their results offered a foundational understanding of refactoring rationale, aligning with empirical research focused on the developer perspective.

Tsantalis et al. [86] conducted a multidimensional empirical investigation of refactoring activity across three widely used open-source projects, such as *JUnit*, *HTTPCore*, and *HTTPClient*, among others. In their study, they combined automated UMLDiff-based refactoring detection with manual qualitative analysis to uncover developer motivations behind applied refactorings. For that, they manually examined hundreds of refactoring observations and identified a motivation taxonomy centered on the topics of code smell resolution, extension-oriented changes, and backward compatibility preservation, among others. Their results revealed that developers frequently refactor not only to improve design quality but also to introduce extension points and maintain stable APIs.

Silva et al. [79] presented the first large-scale empirical study based on the analysis of refactoring operations by collecting hundreds of Java projects from GitHub, and detecting thousands of refactoring operations with the RefactoringMiner tool. Subsequently, they directly asked the developers who performed them to explain their actions. Through thematic analysis consisting of 463 refactorings from 222 commits across 124 software projects, the authors generated a catalogue of 44 distinct motivations concerning 12 refactoring types, revealing that real-world refactoring is primarily driven by evolving requirements such as adding new features, enabling extensions, or supporting API changes. Thus, this study provided strong empirical evidence to build the current ground truth on the fact that developers refactor code to support ongoing maintenance tasks, and not only to fix design flaws.

Palomba et al. [59] investigated 12,922 manually validated refactoring operations across 63 releases from projects *Ant*, *ArgoUML*, and *Xerces-J* to investigate how different types of code changes enhance the application of refactoring operations. By analyzing modifications such as *Fault Repairing*, *Feature Introduction*, and *General maintenance* across 28 refactoring types, they confirmed that refactoring is more often triggered by evolving requirements than by quality issues alone, thus matching previous research efforts [79].

AlOmar et al. [4] studied how developers express their motivation leading them to perform refactoring activities by mining more than 322,000 commit messages related to commits with identified refactoring operations across a set of open-source Java projects. Through their commit message approach, they introduced the concept of *Self-admitted* refactoring to capture explicit developer-documented refactoring actions. Their findings show that developers do frequently document the reasons for performing refactoring operations with motivations such as *improving abstraction*, *reducing complexity*, and *enhancing readability*, among others. Moreover, their findings reflected that commits with self-admitted refactoring exhibited more refactoring activity than those without, demonstrating that self-affirmed refactoring messages provide a reliable window into the motivations guiding developers when performing refactoring activity.

Paixão et al. [58] empirically investigated how developers perform and reason about their refactoring operations during modern code review by mining 1,780 reviewed code changes from six open-source systems. They manually reviewed discussions and revision histories, thereby classifying developer motivations into seven main categories, such as *feature addition*, *bug fixing*, and *pure refactoring*, among others. Thus, they demonstrated how such motivations lead to the selection, composition, and evolution of 7,259 detected refactoring operations. It is important to note that the authors highlighted the fact that refactorings in code review are rarely isolated, since they are part of iterative, motivation-based compositions that adapt as code reviewers provide feedback.

Recently, Ivers et al. [35] investigated the criteria industry developers use when deciding which specific refactoring changes to apply, to better understand industrial practitioners' perceptions. They conducted ten in-depth practitioner interviews, a large industry survey composed of 142 interviewees, and an analysis of 26 state-of-the-art refactoring recommendation tools. Thereby, the authors identified thirteen main reasons that guide industry-based refactoring choices. These included process-oriented concerns such as *change size*, *cost/time ratio*, and *change proneness*, as well as software quality concerns like *code complexity*, *readability*, and *code scalability*, among others. While the performed survey showed that industry developers highly supported the identified thirteen reasons, the analysis of the refactoring recommendation tools shows that only 5 of the reasons are currently covered by the state-of-the-art in refactoring recommendation. By highlighting this mismatch, the study exposed a significant gap between developer decision-making practices and current automated refactoring recommendation capabilities.

Focusing on the insights hidden behind developers' refactoring motivations, Pantiuchina et al. [60] performed a large-scale study to understand why developers refactor code in open-source projects, complementing previous findings from developer surveys [79]. They analyzed 287,813 refactoring operations across 150 systems, examining the relationship between refactoring operations and process/product metrics. The authors highlighted that a relationship exists between metrics and refactoring operations, along with a detailed taxonomy of refactoring motivations. Our

Table 21. Related work on the existing previous research on Refactoring Recommendation.

	Wang [89]	Tsantalis et al. [86]	Silva et al. [79]	Palomba et al. [59]
Granularity	Human rationale	Class, Method, Package	Refactoring	Class, Method, Package
PL	-	Java	Java	Java
Projects	4	3	124	3
Analysis Scope	Qualitative assessment, Exploratory empirical study, Human rationale analysis	Quantitative and qualitative assessment	Human rationale analysis	Quantitative classitication
Dataset	10 interview transcripts	Own mining	Own mining	Bavota et al. [8]
Model	Empirical grounded-theory model	UMLDiff, Open-Coding, Rule-based classification	Thematic Analysis Model	Logistic regression
Mining tools	-	UMLDiff	RefactoringMiner	REFFINDER
Refactoring types	0	11	12	28
Discussion	Yes	Yes	Yes	Yes
Results	Yes	Yes	Yes	No
Scripts	No	No	Yes	No
Datasets	No	No	Yes	No
	AlOmar et al. [4]	Paixão et al. [58]	Ivers et al. [35]	Pantiuchina et al. [60]
Granularity	Class, Method, Package, Attribute	Class, Method, Attribute	Class, Method, Package	Class, Method, Package, Attribute
PL	Java	Java	-	Java
Projects	3,795	6	-	150
Analysis Scope	Large-scale empirical mining study	Manual investigation on code reviews	Interview-based analysis	Quantitative/Qualitative analysis
Dataset	Own mining	Paixão et al. [57]	Interview transcripts	Own mining
Model	Text mining, manual thematic analysis	Manual classification	Thematic analysis	Mixed-effect logistic regression
Mining tools	RefactoringMiner	RefactoringMiner	-	RefactoringMiner, DECOR, PMD, CK
Refactoring types	16	13	-	15
Discussion	Yes	Yes	Yes	Yes
Results	No	Yes	Yes	Yes
Scripts	No	Yes	No	No
Datasets	No	Yes	Yes	Yes
	Our work			
Granularity	Source Code			
PL	Java			
Projects	114			
Analysis Scope	Descriptive analysis, Qualitative assessment, Taxonomy			
Dataset	Silva et al. [79]			
Model	Marco-o1, Deepseek R1, Mistral NeMo, Phi-4			
Mining tools	RefactoringMiner, PyDriller, CK, CoRed			
Refactoring types	103			
Discussion	Yes			
Results	Yes			
Scripts	Yes			
Datasets	Yes			

study differs from Pantiuchina et al.'s [60] approach in study design and goals. While [60] relied on keyword lookup and the RMT tool for PR analysis, we propose a novel methodology employing Language Model analysis, and therefore following the latest research trends on refactoring analysis. Moreover, while [60] focused on product quality and developer-related metrics, our study prioritizes established product and process metrics. Lastly, while the goal was to correlate product quality and developer metrics with refactoring in PRs, we aim to comprehensively analyze motivations for refactoring across various contexts and define a catalogue of motivations supported by a broader selection of metrics from established guidelines.

7 CONCLUSION

Our study investigated the motivations behind performing code refactoring employing LLMs in version-control history data associated with the refactoring, compared their output with human rationale, and with the referenced prior study [79]. Moreover, we studied the extent to which product and process metrics can describe such RMs by analyzing their feature importance and correlation levels. Our results reveal that while LLMs agree with human judgment in most cases, they often diverge in deeper, architectural reasoning, favouring local, surface-level cues over systemic design intent. Still, LLMs demonstrate value in extending and clarifying RMs, surfacing implicit knowledge that is often under-documented.

We further showed that most LLM-derived RMs are pragmatic, driven by clarity and simplification, rather than architectural ideals. Using SMs to depict developer experience and code readability proved informative but insufficient as standalone signals, reinforcing the need for hybrid solutions.

Future research should focus on developing context-aware, LLM-guided refactoring recommendation systems that integrate both behavioural signals, such as RMs, and project-level insights. Similarly, future work should exploit the potential of further exploring prompting techniques, e.g., using multi-turn prompts, system-level metadata, or dependency graphs as prompt context, to provide LLMs with a more holistic context level. We believe such studies would provide LLMs with a more holistic analysis of the context involving the refactoring operation. LLMs can enhance traceability, documentation, and onboarding when paired with system context and developer interaction data. For researchers, this opens a space to explore enriched prompting, system-aware modeling, and empirical evaluation of hybrid systems. For practitioners, LLMs offer an opportunity to make the "why" behind code changes visible, fostering better collaboration and maintainability. LLMs do not just replicate developer intent; they can help understand, explain, and extend it. The next challenge is building tools that channel this capability with precision, context, and reliability.

DECLARATIONS

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

- **Mikel Robredo:** Writing – Original Draft Preparation, Data Curation, Software. Mikel provided essential data preparation, developed the implementation of the study, contributed to drafting specific sections for the writing of the registered report, and contributed significantly to writing the manuscript.
- **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 analysis of data 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.[8]
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.

REFERENCES

[1] Marah I Abdin, Jyoti Aneja, Harkirat S. Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. 2024. Phi-4 Technical Report. *CoRR* abs/2412.08905 (2024). https://doi.org/10.48550/ARXIV.2412.08905 arXiv:2412.08905

[2] Kamal Acharya, Alvaro Velasquez, and Houbing Herbert Song. 2024. A Survey on Symbolic Knowledge Distillation of Large Language Models. *IEEE Trans. Artif. Intell.* 5, 12 (2024), 5928–5948. https://doi.org/10.1109/TAI.2024.3428519

[3] Pouria Alikhanifard and Nikolaos Tsantalis. 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, Article 40 (Jan. 2025), 63 pages. https://doi.org/10.1145/3696002

[4] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can refactoring be self-affirmed?: an exploratory study on how developers document their refactoring activities in commit messages. In *Proceedings of the 3rd International Workshop on Refactoring, IWOR@ICSE 2019, Montreal, QC, Canada, May 28, 2019*, Nikolaos Tsantalis, Yuanfang Cai, and Serge Demeyer (Eds.). IEEE / ACM, 51–58. https://doi.org/10.1109/IWOR.2019.00017

[5] Theodore W Anderson and Donald A Darling. 1952. Asymptotic theory of certain" goodness of fit" criteria based on stochastic processes. *The annals of mathematical statistics* (1952), 193–212.

[6] Maurício Aniche. 2015. *Java code metrics calculator (CK)*. Available in https://github.com/mauricioaniche/ck/.

[7] Paris Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimäki, Darius Sas, Saulo S. de Toledo, and Angeliki-Agathi Tsintzira. 2021. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Softw.* 38, 3 (2021), 61–71. https://doi.org/10.1109/MS.2020.3024958

[8] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *J. Syst. Softw.* 107 (2015), 1–14. https://doi.org/10.1016/J.JSS.2015.05.024

[9] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)* 57, 1 (1995), 289–300.

[10] Patrick Billingsley. 2013. *Convergence of probability measures.* John Wiley & Sons.

[11] Albert H Bowker. 1948. A test for symmetry in contingency tables. *Journal of the american statistical association* 43, 244 (1948), 572–574.

[12] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

[13] Leo Breiman, Jerome Friedman, Charles J. Stone, and R.A. Olshen. 1984. *Classification and regression trees Regression trees.*

[14] George Casella and Roger Berger. 2024. *Statistical inference.* CRC press.

[15] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 785–794. https://doi.org/10.1145/2939672.2939785

[16] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 785–794. https://doi.org/10.1145/2939672.2939785

[8]https://doi.org/10.5281/zenodo.15209154

[17] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[18] Cyril Dalmasso, Philippe Broet, and Thierry Moreau. 2005. A simple procedure for estimating the false discovery rate. *Bioinformatics* 21, 5 (2005), 660–668.

[19] CP Dancey. 2007. *Statistics without maths for psychology.* Prentice Hall.

[20] Joshua Davis, Liesbet Van Bulck, Brigitte N Durieux, and Charlotta Lindvall. 2024. The Temperature Feature of ChatGPT: Modifying Creativity for Clinical Research. *JMIR Hum Factors* 11 (8 Mar 2024), e53559. https://doi.org/10.2196/53559

[21] Kayla Depalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. 2024. Exploring ChatGPT's code refactoring capabilities: An empirical study. *Expert Syst. Appl.* 249 (2024), 123602. https://doi.org/10.1016/J.ESWA.2024.123602

[22] Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252.

[23] Matteo Esposito and Davide Falessi. 2023. Uncovering the Hidden Risks: The Importance of Predicting Bugginess in Untouched Methods. In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2023, Bogotá, Colombia, October 2-3, 2023*, Leon Moonen, Christian D. Newman, and Alessandra Gorla (Eds.). IEEE, 277–282. https://doi.org/10.1109/SCAM59687.2023.00039

[24] Matteo Esposito, Xiaozhou Li, Sergio Moreschini, Noman Ahmad, Tomás Cerný, Karthik Vaidhyanathan, Valentina Lenarduzzi, and Davide Taibi. 2025. Generative AI for Software Architecture. Applications, Trends, Challenges, and Future Directions. *CoRR* abs/2503.13310 (2025). https://doi.org/10.48550/ARXIV.2503.13310 arXiv:2503.13310

[25] Matteo Esposito, Francesco Palagiano, Valentina Lenarduzzi, and Davide Taibi. 2024. Beyond Words: On Large Language Models Actionability in Mission-Critical Risk Analysis. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024*, Xavier Franch, Maya Daneva, Silverio Martínez-Fernández, and Luigi Quaranta (Eds.). ACM, 517–527. https://doi.org/10.1145/3674805.3695401

[26] Matteo Esposito, Francesco Palagiano, Valentina Lenarduzzi, and Davide Taibi. 2025. On Large Language Models in Mission-Critical IT Governance: Are We Ready Yet?. In *47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2025, Ottawa, ON, Canada, April 27 - May 3, 2025*. IEEE, 504–515. https://doi.org/10.1109/ICSE-SEIP66354.2025.00050

[27] Matteo Esposito, Mikel Robredo, Murali Sridharan, Guilherme Horta Travassos, Rafael Peñaloza, and Valentina Lenarduzzi. 2025. A Call for Critically Rethinking and Reforming Data Analysis in Empirical Software Engineering. *CoRR* abs/2501.12728 (2025). https://doi.org/10.48550/ARXIV.2501.12728 arXiv:2501.12728

[28] Davide Falessi, Simone Mesiano Laureani, Jonida Çarka, Matteo Esposito, and Daniel Alencar da Costa. 2023. Enhancing the defectiveness prediction of methods and classes via JIT. *Empir. Softw. Eng.* 28, 2 (2023), 37. https://doi.org/10.1007/S10664-022-10261-Z

[29] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 31–53. https://doi.org/10.1109/ICSE-FOSE59343.2023.00008

[30] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[31] Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley. http://martinfowler.com/books/refactoring.html

[32] Jun Gao, Yun Peng, and Xiaoxue Ren. 2025. \texttt{ReMind}: Understanding Deductive Code Reasoning in LLMs. *CoRR* abs/2511.00488 (2025). https://doi.org/10.48550/ARXIV.2511.00488 arXiv:2511.00488

[33] Hong Han, Xiaoling Guo, and Hua Yu. 2016. Variable selection using mean decrease accuracy and mean decrease gini based on random forest. In *International conference on software engineering and service science (icsess)*. IEEE, 219–224.

[34] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[35] James Ivers, Anwar Ghammam, Khouloud Gaaloul, Ipek Ozkaya, Marouane Kessentini, and Wajdi Aljedaani. 2024. Mind the Gap: The Disconnect Between Refactoring Criteria Used in Industry and Refactoring Recommendation Tools. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024, Flagstaff, AZ, USA, October 6-11, 2024*. IEEE, 138–150. https://doi.org/10.1109/ICSME58944.2024.00023

[36] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.* 39, 6 (2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[37] Dae-Kyoo Kim. 2025. Comparative analysis of design pattern implementation validity in LLM-based code refactoring. *J. Syst. Softw.* 230 (2025), 112519. https://doi.org/10.1016/J.JSS.2025.112519

[38] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12,*

Cary, NC, USA - November 11 - 16, 2012, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 50. https://doi.org/10.1145/2393596.2393655

[39] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[40] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 315–322. https://doi.org/10.1145/3387940.3392191

[41] Guilherme Lacerda, Fábio Petrillo, Marcelo Pimenta, and Yann-Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *J. Syst. Softw.* 167 (2020), 110610. https://doi.org/10.1016/J.JSS.2020.110610

[42] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* (1977), 159–174.

[43] Valentina Lenarduzzi, Terese Besker, Davide Taibi, Antonio Martini, and Francesca Arcelli Fontana. 2021. A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. *J. Syst. Softw.* 171 (2021), 110827. https://doi.org/10.1016/J.JSS.2020.110827

[44] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimäki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *J. Syst. Softw.* 198 (2023), 111575. https://doi.org/10.1016/J.JSS.2022.111575

[45] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study. *Autom. Softw. Eng.* 32, 1 (2025), 26. https://doi.org/10.1007/S10515-025-00500-0

[46] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[47] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. 2019. Ablation Studies in Artificial Neural Networks. *CoRR* abs/1901.08644 (2019). arXiv:1901.08644 http://arxiv.org/abs/1901.08644

[48] Alessandro Midolo and Emiliano Tramontana. 2025. Refactoring Loops in the Era of LLMs: A Comprehensive Study. *Future Internet* 17, 9 (2025). https://doi.org/10.3390/fi17090418

[49] Prabhaker Mishra, Chandra M Pandey, Uttam Singh, Anshul Gupta, Chinmoy Sahu, and Amit Keshri. 2019. Descriptive statistics and normality tests for statistical data. *Annals of cardiac anaesthesia* 22, 1 (2019), 67–72.

[50] Mistral AI team Mistral AI. 2023. Mistral 7B. https://mistral.ai/news/announcing-mistral-7b

[51] Mistral AI team Mistral AI. 2024. Mistral Nemo. https://mistral.ai/news/mistral-nemo

[52] I. C. Mogotsi. 2010. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze: Introduction to information retrieval - Cambridge University Press, Cambridge, England, 2008, 482 pp, ISBN: 978-0-521-86571-5. *Inf. Retr.* 13, 2 (2010), 192–195. https://doi.org/10.1007/S10791-009-9115-Y

[53] Alexey Natekin and Alois C. Knoll. 2013. Gradient boosting machines, a tutorial. *Frontiers Neurorobotics* 7 (2013), 21. https://doi.org/10.3389/FNBOT.2013.00021

[54] Kristin K. Nicodemus. 2011. Letter to the Editor: On the stability and ranking of predictors from random forest variable importance measures. *Briefings Bioinform.* 12, 4 (2011), 369–373. https://doi.org/10.1093/BIB/BBR016

[55] Ally S. Nyamawe, Hui Liu, Zhendong Niu, Wentao Wang, and Nan Niu. 2018. Recommending Refactoring Solutions Based on Traceability and Code Metrics. *IEEE Access* 6 (2018), 49460–49475. https://doi.org/10.1109/ACCESS.2018.2868990

[56] OpenAI. 2025. openai — pypi.org. https://pypi.org/project/openai/. [Accessed 28-04-2025].

[57] Matheus Paixão, Jens Krinke, DongGyun Han, and Mark Harman. 2018. CROP: linking code reviews to source code changes. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 46–49. https://doi.org/10.1145/3196398.3196466

[58] Matheus Paixão, Anderson G. Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. Behind the Intents: An In-depth Empirical Study on Software Refactoring in Modern Code Review. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 125–136. https://doi.org/10.1145/3379597.3387475

[59] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, Giuseppe Scanniello, David Lo, and Alexander Serebrenik (Eds.). IEEE Computer Society, 176–185. https://doi.org/10.1109/ICPC.2017.38

[60] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why Developers Refactor Source Code: A Mining-based Study. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 29:1–29:30. https://doi.org/10.1145/3408302

[61] L. Pascarella, F. Palomba, and A. Bacchelli. 2018. Re-evaluating method-level bug prediction. In *International Conference on Software Analysis, Evolution and Reengineering (SANER).* 592–601.

[62] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. How do i refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empir. Softw. Eng.* 27, 1 (2022), 11. https://doi.org/10.1007/S10664-021-10045-X

[63] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 582–586. https://doi.org/10.1145/3663529.3663803

[64] Farhad Pourpanah, Moloud Abdar, Yuxuan Luo, Xinlei Zhou, Ran Wang, Chee Peng Lim, Xi-Zhao Wang, and Q. M. Jonathan Wu. 2023. A Review of Generalized Zero-Shot Learning Methods. *IEEE Trans. Pattern Anal. Mach. Intell.* 45, 4 (2023), 4051–4070. https://doi.org/10.1109/TPAMI.2022.3191696

[65] Foyzur Rahman and Premkumar T. Devanbu. 2013. How, and why, process metrics are better. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 432–441. https://doi.org/10.1109/ICSE.2013.6606589

[66] Mohammad Masudur Rahman and Chanchal K. Roy. 2014. An insight into the pull requests of GitHub. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.). ACM, 364–367. https://doi.org/10.1145/2597073.2597121

[67] Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo Ansaldi Oliva, Yasutaka Kamei, and Ahmed E. Hassan. 2022. The Impact of Feature Importance Methods on the Interpretation of Defect Classifiers. *IEEE Trans. Software Eng.* 48, 7 (2022), 2245–2261. https://doi.org/10.1109/TSE.2021.3056941

[68] Mikel Robredo, Matteo Esposito, Fabio Palomba, Rafael Peñaloza, and Valentina Lenarduzzi. 2024. Analyzing the Ripple Effects of Refactoring. *A Registered Report. This Registered Report has been accepted at ICSME* (2024).

[69] Mikel Robredo, Nyyti Saarimäki, Matteo Esposito, Davide Taibi, Rafael Peñaloza, and Valentina Lenarduzzi. 2025. Evaluating time-dependent methods and seasonal effects in code technical debt prediction. *J. Syst. Softw.* 230 (2025), 112545. https://doi.org/10.1016/J.JSS.2025.112545

[70] Nyyti Saarimäki, Sergio Moreschini, Francesco Lomio, Rafael Peñaloza, and Valentina Lenarduzzi. 2022. Towards a Robust Approach to Analyze Time-Dependent Data in Software Engineering. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022.* IEEE, 36–40. https://doi.org/10.1109/SANER53432.2022.00015

[71] Nyyti Saarimäki, Mikel Robredo, Valentina Lenarduzzi, Sira Vegas, Natalia Juristo, and Davide Taibi. 2025. Does microservice adoption impact the velocity? A cohort study. *Empir. Softw. Eng.* 30, 5 (2025), 130. https://doi.org/10.1007/S10664-025-10673-7

[72] Claude Sammut and Geoffrey I. Webb (Eds.). 2017. *Encyclopedia of Machine Learning and Data Mining.* Springer. https://doi.org/10.1007/978-1-4899-7687-1

[73] Margarete Sandelowski. 1995. Sample size in qualitative research. *Research in nursing & health* 18, 2 (1995), 179–183.

[74] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *J. Softw. Evol. Process.* 30, 6 (2018). https://doi.org/10.1002/SMR.1958

[75] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Inf. Softw. Technol.* 178 (2025), 107610. https://doi.org/10.1016/J.INFSOF.2024.107610

[76] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.

[77] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring Programs Using Large Language Models with Few-Shot Examples. In *30th Asia-Pacific Software Engineering Conference, APSEC 2023, Seoul, Republic of Korea, December 4-7, 2023.* IEEE, 151–160. https://doi.org/10.1109/APSEC60848.2023.00025

[78] Danilo Silva, João Paulo da Silva, Gustavo Jansen de Souza Santos, Ricardo Terra, and Marco Túlio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Trans. Software Eng.* 47, 12 (2021), 2786–2802. https://doi.org/10.1109/TSE.2020.2968072

[79] Danilo Silva, Nikolaos Tsantalis, and Marco Túlio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 858–870. https://doi.org/10.1145/2950290.2950305

[80] Minjun Son, Yun-Jae Won, and Sungjin Lee. 2025. Optimizing Large Language Models: A Deep Dive into Effective Prompt Engineering Techniques. *Applied Sciences* 15, 3 (2025). https://doi.org/10.3390/app15031430

[81] Charles Spearman. 1961. The proof and measurement of association between two things. *The American Journal of Psychology* (1961).

[82] M. A. Stephens. 1974. EDF Statistics for Goodness of Fit and Some Comparisons. *J. Amer. Statist. Assoc.* 69, 347 (1974), 730–737.

[83] Tom Stuart. 2015. *Understanding computation - from simple machines to impossible programs.* O'Reilly. http://www.oreilly.de/catalog/9781449329273/index.html

[84] Gábor Szoke, Gabor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014.* IEEE Computer Society, 95–104. https://doi.org/10.1109/SCAM.2014.18

[85] Gábor Szoke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality. In *Computational Science and Its Applications - ICCSA 2014 - 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part V (Lecture Notes in Computer Science)*, Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, Carmelo Maria Torre, Jorge Gustavo Rocha, Maria Irene Falcão, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi (Eds.), Vol. 8583. Springer, 524–540. https://doi.org/10.1007/978-3-319-09156-3_37

[86] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *Center for Advanced Studies on Collaborative Research, CASCON '13, Toronto, ON, Canada, November 18-20, 2013*, James R. Cordy, Krzystof Czarnecki, and Sang-Ah Han (Eds.). IBM / ACM, 132–146. http://dl.acm.org/citation.cfm?id=2555539

[87] Dimitrios Tsoukalas, Nikolaos Mittas, Alexander Chatzigeorgiou, Dionisis D. Kehagias, Apostolos Ampatzoglou, Theodoros Amanatidis, and Lefteris Angelis. 2022. Machine Learning for Technical Debt Identification. *IEEE Trans. Software Eng.* 48, 12 (2022), 4892–4906. https://doi.org/10.1109/TSE.2021.3129355

[88] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[89] Yi Wang. 2009. What motivate software engineers to refactor source code? evidences from professional developers. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada.* IEEE Computer Society, 413–416. https://doi.org/10.1109/ICSM.2009.5306290

[90] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[91] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

[92] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering.* Springer. https://doi.org/10.1007/978-3-642-29044-2

[93] S Paul Wright. 1992. Adjusted p-values for simultaneous inference. *Biometrics* (1992), 1005–1013.

[94] Yue Yu, Huaimin Wang, Gang Yin, and Charles X. Ling. 2014. Reviewer Recommender of Pull-Requests in GitHub. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014.* IEEE Computer Society, 609–612. https://doi.org/10.1109/ICSME.2014.107

[95] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Inf. Softw. Technol.* 74 (2016), 204–218. https://doi.org/10.1016/J.INFSOF.2016.01.004

[96] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. *CoRR* abs/2312.15223 (2023). https://doi.org/10.48550/ARXIV.2312.15223 arXiv:2312.15223

[97] Zejun Zhang, Zhenchang Xing, Xiaoxue Ren, Qinghua Lu, and Xiwei Xu. 2024. Refactoring to Pythonic Idioms: A Hybrid Knowledge-Driven Approach Leveraging Large Language Models. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1107–1128. https://doi.org/10.1145/3643776

[98] Yu Zhao, Huifeng Yin, Bo Zeng, Hao Wang, Tianqi Shi, Chenyang Lyu, Longyue Wang, Weihua Luo, and Kaifu Zhang. 2024. Marco-o1: Towards Open Reasoning Models for Open-Ended Solutions. *CoRR* abs/2411.14405 (2024). https://doi.org/10.48550/ARXIV.2411.14405 arXiv:2411.14405

[99] Tianyang Zhong, Zhengliang Liu, Yi Pan, Yutong Zhang, Yifan Zhou, Shizhe Liang, Zihao Wu, Yanjun Lyu, Peng Shu, Xiaowei Yu, Chao Cao, Hanqi Jiang, Hanxu Chen, Yiwei Li, Junhao Chen, Huawen Hu, Yihen Liu, Huaqin Zhao,

Shaochen Xu, Haixing Dai, Lin Zhao, Ruidong Zhang, Wei Zhao, Zhenyuan Yang, Jingyuan Chen, Peilong Wang, Wei Ruan, Hui Wang, Huan Zhao, Jing Zhang, Yiming Ren, Shihuan Qin, Tong Chen, Jiaxi Li, Arif Hassan Zidan, Afrar Jahin, Minheng Chen, Sichen Xia, Jason Holmes, Yan Zhuang, Jiaqi Wang, Bochen Xu, Weiran Xia, Jichao Yu, Kaibo Tang, Yaxuan Yang, Bolun Sun, Tao Yang, Guoyu Lu, Xianqiao Wang, Lilong Chai, He Li, Jin Lu, Lichao Sun, Xin Zhang, Bao Ge, Xintao Hu, Lian Zhang, Hua Zhou, Lu Zhang, Shu Zhang, Ninghao Liu, Bei Jiang, Linglong Kong, Zhen Xiang, Yudan Ren, Jun Liu, Xi Jiang, Yu Bao, Wei Zhang, Xiang Li, Gang Li, Wei Liu, Dinggang Shen, Andrea Sikora, Xiaoming Zhai, Dajiang Zhu, and Tianming Liu. 2024. Evaluation of OpenAI o1: Opportunities and Challenges of AGI. *CoRR* abs/2409.18486 (2024). https://doi.org/10.48550/ARXIV.2409.18486 arXiv:2409.18486

APPENDIX

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

A SUPPLEMENTARY MATERIAL FOR THE DATA COLLECTION

A.1 Expanded description on the refactoring collection using RMT

RMT remains the state-of-the-art refactoring mining tool as it has a refactoring detection accuracy close to 100%, [9] which overcomes the capabilities of other existing tools such as RefDiff [78]). RMT 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. Table 2 shows the categorization of the 103 refactorings of our study according to their types as defined by Fowler [31]. Out of the RTs that can be detected, only a subset of the detected RTs is classified in the original Fowler catalogue. Nonetheless, the RMT can also identify types that cannot be mapped to the original catalogue, such as the composite refactoring *Move and Inline Method*.

We conducted the mining process with RMT on a workstation equipped with an Intel Core i9-13900KF CPU (24 cores, 32 threads), 64GB of RAM, and 7.27TB of disk storage. Similarly, we configured RMT to use a maximum share of RAM of 57GB, allowing other processes to continue running smoothly. Leveraging the custom options RMT offers, we launched the tool for each project to report the refactoring activity as a JSON file comprising each of the analyzed commits, the set of applied refactoring operations, as well as the classes/methods subject to them. Yet, projects *liferay/liferay-plugins*, *checkstyle/checkstyle*, *deeplearning4j/deeplearning4j*, and *jOOQ/jOOQ* resulted in exceeding *heap-space* error, which we could not resolve given our available computational resources. In addition, projects *jersey/jersey*, *crate/crate*, *JetBrains/MPS*, *siacs/Conversations*, *kuujo/-copycat*, *bitfireAT/davroid* resulted in untraceable stalled processes. Consequently, after a prolonged period of multiple attempts, we opted to proceed without the mentioned projects. The discarded projects accounted for a total of 13 refactoring commits present in the reference study. However, since we could collect the SMs related to affected refactoring commits, and the replication package from the reference study provided the affected RT, we still considered the affected commits, and therefore their ground-truth refactoring motivations, in the study. We acknowledge the impact of this decision as a threat to validity in Section 5.

A.2 Expanded description on the conducted sampling strategy

Phase 1: Initially, we aimed to achieve enough representativeness for each of the mined projects and all the RTs with the drawn sample. For such a goal, we decided on setting a data structure to record the sampled refactoring observations, until all the projects had a minimum total of **3 refactorings**. Likewise, all the refactoring types had a minimum representation of the same number of refactorings on a global level.

Phase 2: The previous phase did not reach the full representativeness of the mined 114 projects, as some projects did not detect refactorings from the RTs left to be sampled. Therefore, we expanded the sampling logic implemented in the previous phase and identified which projects remained under-sampled. We adopted the **reservoir sampling** technique to randomly pick the remaining refactorings [88] to fill the under-sampled projects. Given that we knew the total number n of refactorings for each of the projects remaining unrepresented, and we also knew the k amount of sampled refactoring needed in each of the projects, reservoir sampling ensured that every

[9] https://github.com/tsantalis/RefactoringMiner

refactoring observation within the set of mined refactorings per project had the same probability of being included on the set or *reservoir* of refactorings still pending to be sampled per project.

Phase 3: Phases I and II resulted in a randomly chosen, yet balanced sample of 342 refactoring observations. Therefore, to ensure the target sample size to provide statistically significant results from the data analysis, we pooled all the mined refactorings together and performed pure random sampling with the remaining 43 refactoring observations.

B SUPPLEMENTARY MATERIAL FOR THE DATA ANALYSIS

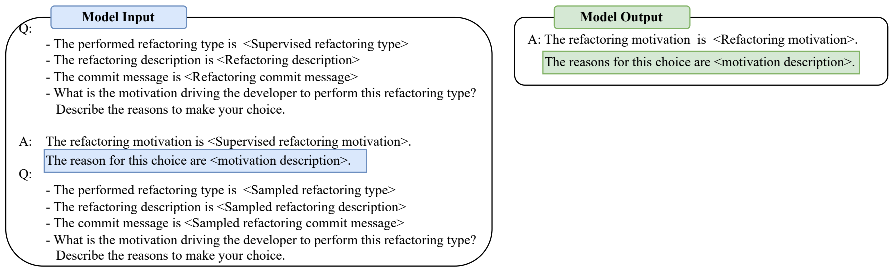
B.1 Selected LLM models

To balance performance and efficiency, we selected high-performing *distilled* LLMs [2]. Model distillation refers to the process of transferring knowledge from an LLM to a smaller one, aiming to preserve performance while reducing model complexity and computational cost (see Section 5). It is a complex task to know which model configuration leads to the most efficient answers when performing prompt engineering [80]. For this reason, we fixed the model temperature at 0.8, which provides a balance between creativity and coherence [20].

- **Marco-o1** [98]: A 7.6B parameter distilled version of OpenAI's o-1 model [99], fine-tuned on curated CoT datasets. It uses Monte Carlo Tree Search (MCTS) and softmax-based scoring to explore reasoning paths, and shows strong performance across math, coding, and logic tasks.[10] The model configuration when performing the prompting included a context length of 4096 tokens.
- **Mistral NeMo** [51]: A 12B model distilled from Mistral 7B [50], optimized for multi-turn reasoning, code generation, and instruction following. It benefits from alignment fine-tuning and supports 128K token inputs.[11]
- **DeepSeek R1**: A 14B distilled model from the DeepSeek-V3-Base series, trained with CoT examples and refined via human feedback and reinforcement learning. It matches o-1-level performance on reasoning benchmarks.[12] Given the bigger size of DeepSeek R1, we configured the context length of this model to be double the one from the previously adopted distilled models, i.e., 8129 tokens.
- **Microsoft Phi-4** [1]: A 14B model with a 16K token context window, trained on high-quality academic and technical data. It consistently outperforms larger models like GPT-4o in formal reasoning tasks.[13] As with DeepSeek R1, given that the size of Phi-4 was the same, we configured the context length limit to be the same, fixed at 8129 tokens.

To run the selected models locally, we used LM Studio, a desktop application designed for experimenting with LLMs in an offline environment.[14] LM Studio integrates directly with the *Hugging Face* model hub, allowing us to download and launch a variety of models easily.[15] It also sets up a local server that mimics the OpenAI API interface, enabling smooth compatibility with tools built around OpenAI's endpoints.[16] For interaction and prompt execution, we relied on the Python `openai` library developed by OpenAI [56]. All experiments were conducted on a high-performance workstation equipped with an NVIDIA GeForce RTX 4090 GPU.

[10] https://huggingface.co/AIDC-AI/Marco-o1
[11] https://huggingface.co/mistralai/Mistral-Nemo-Base-2407 The model configuration when performing the prompting included a context length of 4096 tokens.
[12] https://deepseek.ai
[13] https://huggingface.co/microsoft/phi-4
[14] https://lmstudio.ai
[15] https://huggingface.co/models
[16] https://platform.openai.com/docs/api-reference/introduction


Model Input

Q:
- The performed refactoring type is <Supervised refactoring type>
- The refactoring description is <Refactoring description>
- The commit message is <Refactoring commit message>
- What is the motivation driving the developer to perform this refactoring type? Describe the reasons to make your choice.

A: The refactoring motivation is <Supervised refactoring motivation>.
The reason for this choice are <motivation description>.

Q:
- The performed refactoring type is <Sampled refactoring type>
- The refactoring description is <Sampled refactoring description>
- The commit message is <Sampled refactoring commit message>
- What is the motivation driving the developer to perform this refactoring type? Describe the reasons to make your choice.

Model Output

A: The refactoring motivation is <Refactoring motivation>.
The reasons for this choice are <motivation description>.

Fig. 15. Example of Zero-Shot CoT prompt based on Kojima et al. [39].

B.2 On the adopted CoT prompting strategy

CoT prompting, is a method that enhances the ability of LLMs to perform complex reasoning by eliciting a series of intermediate reasoning steps [90]. CoT involves generating a "chain of thought", that is, a coherent sequence of natural language reasoning steps that lead to the final answer for a problem. It enables models to decompose multi-step problems, suggesting how they arrive at a particular answer. The process mimics a step-by-step thought process, similar to how humans solve complicated reasoning tasks. There are two primary ways to execute: Few-shot CoT prompting [90] and Zero-shot CoT prompting [39]. In our work, we adopted the latter since we did not want to introduce bias in the LLMs' responses based on the provided sample answers, and the ground-truth motivations extracted from the reference work did not include the entirety of the studied RTs. Figure 15 provides an example pseudo-prompt following the Zero-shot CoT prompt strategy. Based on the required outputs for each of the defined RQs, we fine-tuned system and user prompt messages correspondingly, and made them available in the replication package (see Data Availability Statement). The last format of the shared prompt templates, and therefore the prompt messages used in the study, are the result of a progressive fine-tuning process where the quality of the LLM responses was discussed among the authors until a unanimous agreement was found. Thus, the prompt with the user messages we used to feed the LRM was structured as follows:

Structured JSON response. The expected model output consisted of a structured JSON response containing: (i) a brief answer/label with the requested answer, (ii) a concise `answer description`, and (iii) the underlying `reasoning` process involved in building the answer. This process was modified for each task prompting stage, correspondingly, based on each RQ. We instructed the LLMs to present their response in the defined JSON output. For that, we leveraged OpenAI's REST API[17] to enforce the prompted model to process the response output in JSON format.

B.3 On the conducted Open-Coding.

Since we already had the validated RMs, this time we tasked models V1, V2, and V3 to perform open-coding on the 385 RMs. Each model received the detected RM, a description of the RM (generated by the best-performing model from a previous task), and an initially empty list of categories (called the **pooled list**) that could grow during the process. It is important to note that, following the design of an open-coding process, the pool of motivation categories would start empty and greedily increase as more refactoring motivation observations are processed by the validation assistant models.

[17] https://platform.openai.com/docs/guides/structured-outputs

Each model was instructed to assign a clear motivation category (RMC) to the RM and explain why. The models were encouraged to reuse existing categories when possible, but they could also introduce new ones if needed. After V1 and V2 completed their coding, V3 reviewed the same RM and the two proposed categories. V3 then chose the final category based on three options:

(1) If V1 and V2 agreed, V3 accepted that shared category.
(2) If they disagreed, V3 picked one of the two based on their explanations.
(3) If neither fit well, V3 could suggest a new category, with justification.

Likewise, given the absence of existing refactoring motivation categories and to respect the nature of an open-coding task, we followed the zero-shot CoT learning strategy. Each of the models was instructed to consider the already existing categories provided in the user message. Yet, they were allowed to suggest a new motivation category if the existing categories in the list above did not fit the nature of the concerning motivation.

Finally, the human experts focused on a statistically representative sample (95% confidence level, 5% margin of error) [73]. The experts also repeated the open-coding process on the ground-truth dataset[79] to compare and uncover additional developer motivations.

C LIST OF ABBREVIATIONS FROM THE DETECTED REFACTORING TYPES.

Table C.1. List of abbreviations for the Refactoring Types analyzed in the study.

Refactoring Type	Abbr.	Refactoring Type	Abbr.	Refactoring Type	Abbr.
Extract Variable	EV	Change Attribute Access Modifier	CAAM	Remove Attribute Annotation	RAA
Change Parameter Type	CPT	Extract And Move Method	EAMM	Add Thrown Exception Type	ATET
Rename Parameter	RenP	Remove Attribute Modifier	RAM	Encapsulate Attribute	EnA
Extract Method	EM	Change Return Type	CRT	Add Variable Modifier	AVM
Change Variable Type	CVT	Extract Interface	EI	Move And Inline Method	MAIM
Add Parameter	AP	Rename Variable	RV	Inline Method	IM
Add Class Annotation	ACA	Modify Parameter Annotation	MPA	Reorder Parameter	RParam
Remove Method Annotation	RMA	Modify Attribute Annotation	MAA	Inline Attribute	IA
Rename Attribute	RA	Add Attribute Annotation	AAA	Rename Class	RC
Add Parameter Annotation	APA	Parameterize Variable	PV	Parameterize Attribute	PA
Inline Variable	IV	Modify Method Annotation	MMA	Localize Parameter	LP
Merge Parameter	MParam	Remove Class Annotation	RCA	Remove Parameter Annotation	RPA
Merge Attribute	MerA	Change Class Access Modifier	CCAM	Remove Variable Annotation	RVA
Add Method Annotation	AMA	Change Method Access Modifier	CMAM	Change Thrown Exception Type	CTET
Move Method	MM	Add Class Modifier	ACM	Pull Up Method	PUM
Move Attribute	MA	Replace Loop With Pipeline	RLWP	Move Source Folder	MSF
Add Attribute Modifier	AAM	Move Class	MovC	Extract Superclass	ESup
Extract Class	EC	Modify Class Annotation	MCA	Replace Attribute With Variable	RAWV
Replace Generic With Diamond	RGWD	Merge Conditional	MCon	Merge Class	MerC
Invert Condition	IC	Add Method Modifier	AMM	Move And Rename Class	MARM
Replace Variable With Attribute	RVWA	Replace Anonymous With Lambda	RAWL	Remove Variable Modifier	RVM
Rename Method	RM	Remove Thrown Exception Type	RTET	Move Package	MP
Move And Rename Attribute	MARA	Remove Class Modifier	RCM	Rename Package	RPack
Remove Parameter	RemP	Remove Method Modifier	RMM	Split Parameter	SP

Refactoring Type	**Abbr.**	**Refactoring Type**	**Abbr.**	**Refactoring Type**	**Abbr.**
Extract Attribute	ExA	Move And Rename Method	MARM	Split Attribute	SA
Change Attribute Type	CAT	Move Code	MCode	Pull Up Attribute	PUA
Push Down Method	PDM	Split Method	SM	Merge Variable	MV
Split Package	SP	Split Conditional	SC	Replace Attribute	RA
Merge Package	MPack	Split Variable	SV	Replace Anonymous With Class	RAWC
Merge Method	MerM	Split Class	SClass	Replace Pipeline With Loop	RPWL
Change Type Declaration Kind	CTDK	Replace Conditional With Ternary	RCWT	Try With Resources	TWR
Remove Parameter Modifier	RPM	Push Down Attribute	PDA	Assert Throws	AT
Add Parameter Modifier	APM	Add Variable Annotation	AVA	Merge Catch	MCat
Collapse Hierarchy	CH	Extract Subclass	ESub	Modify Variable Annotation	MVA
Parameterize Test	PT				

(**Abbr.**: Abbreviation)

D LIST OF REFACTORING MOTIVATION CATEGORIES PER MOTIVATION FAMILY, ALONG WITH THEIR DUPLICATE FREQUENCIES.

Table D.1. List of unique motivation categories and their respective frequencies grouped into the defined RMCs.

Motivation Category	**Unique motivations (# Occurrences)**
Code Clarity and Readability	Enhance (2) / Improve (2) **Code Clarity** (11) and Simplification (12) / and Maintainability (2) / and Abstraction (11), Improve Readability (3) / Data Structure Semantics (2), Enhance (12) / Improve (1) **Code Readability and Maintainability** (10), Enhance (3) **Code Readability** (1) and Consistency (4) / and Simplification (1) / by Isolating Logic (1), Enhance (3) **Code Organization** (6) and Promote Reusability (1), **Annotation Update for Deprecation** (2), **Package Organization Alignment** (3), **Ensure Correctness and Consistency** (2), **Clarify** Nullability Semantics (3) / Class Purpose (1), **Enhance Readability** (6) and Maintainability (11), **Improve Clarity by Isolating Complex Expressions** (1), **Enhance Code Conciseness and Readability** (1)
Maintainability and Modularity	Enhance Modularity (1) / and Reusability (14) / Code Flexibility and Maintainability (1) / Code Modularity (1), Promote Reuse (3), Improve Code Organization (1), Increase Flexibility (7), Improve Code Quality (1) Improvement (1), Centralize Shared Behavior (1), Reorganize Code Structure (2), Leverage Existing Implementation (1)
Encapsulation and Abstraction	Increase (1) / Improve (1) / Enhance (2) **Encapsulation Breach** (1) / Enhancement (15), **Attribute Coupling Reduction** (1), **Encapsulate Attribute** (1), Reduce Global State and Enhance Encapsulation (1) / Class Coupling and Enhance Encapsulation (1)
Testing and Reliability	**Reusability of Test Methods** (2), **Enhance Test Organization** (2), **Test Isolation Improvement** (2), **Reduce Coupling and Enhance Testability** (2), Enable Test Flexibility (3) / Test Execution (1), **Move Method to Facilitate Test Setup** (1), **Enhance Test Reliability** (1), **Adopt Modern Testing Framework** (2), **Eliminate Unnecessary Variables** (3)
Code Simplification and Redundancy Reduction	**Reduce Redundancy** (9), **Simplify Code** (4) by Reducing Complexity (4) / Structure (1) / by Removing Unnecessary Annotations (1) / by Removing Unnecessary Elements (1) / Simplify Code by Reducing Redundancy (1), **Eliminate Redundant Code** (1), **Remove Unnecessary Code Elements** (4) / Redundant Parameters (1), Redundant Annotation (1) / Unnecessary Variable (1), **Simplify Parameter Handling by Reducing Argument Count** (1), **Reduce Code Duplication** (7) and Enhance Flexibility (1), **Simplify Constructor Logic** (1), **Reduce Complexity** (16) by Removing Unnecessary Code Segments (2) / by Eliminating Unnecessary Abstractions (1), **Eliminate Misleading or Unnecessary Information** (1), **Simplify Constant Management** (1), **Eliminate Redundancy** (2), **Reduce Duplication** (1) / Local Complexity (1), **Simplify Parameter Handling** (4), **Simplify Suppression Warnings by Reducing the Number of Types Targeted** (1), **Simplify Dependency Management** (1), **Reduce Code Duplication and Enhance Flexibility** (1), **Simplify Class Structure** (2), **Remove Unnecessary Checked Exception** (1), **Reduce Dependencies and Simplify State Management** (1), **Eliminate Unnecessary Suppression** (2) / Variables (2) / Restrictions (1)
Exception and Error Handling	**Correctness** (1), **Change Thrown Exception Type** (1), **Assert Exception Handling** (1), **Improve Error Handling** (4), **Clarify Exception Semantics** (1), **Ensure Correctness of Method Overrides** (2), **Remove Unnecessary Checked Exception** (1), **Ensure Correctness of Exception Handling** (1), **Improve Exception Handling Specificity** (1)
Type and Parameter Handling	**Type Safety Enhancement** (7), **Change Parameter Type** (2), Clarify Parameter Semantics (1) / Exception Semantics (1), Improve type specificity (1), Enhance Parameter Clarity and Specificity (1)
Structural Reorganization	Move Class (1) / Feature Module (1), **Split Package** (1), **Rename and Reclassification** (1), Consolidate (1), Package Structure Optimization (1) / Consolidation (1), **Move and Rename Attribute** (1), **Extract Interface** (1), **Simplify Hierarchy** (1), **Improve Specificity in Class Responsibilities** (1)

Motivation Category	**Unique motivations (# Occurrences)**
Performance and Resource Management	**Performance Optimization** (1), **Facilitate Thread Management** (1), Enhance Memory Management (2) / Logging and Integration (1), **Resource Management Optimization** (2), **Performance Enhancement** (1)
Consistency and Standardization	**Consistency and Metric Accuracy** (1), **Annotation Update for Deprecation** (3), **Improve Consistency** (1), **Ensure Correctness** (3), **Replace Custom Abstractions with Standard Methods** (1), **Address Annotation Warnings** (1)
Design Principles and Patterns	**Enforce Design Patterns** (1), **Apply SRP** (1), **Refactor for Code Compatibility** (1), **Enhance Interface Abstraction** (1), **Improve Internal Cohesion** (1)
Security and Safety	**Ensure Null Safety** (1) / Thread Safety (1), **Security and Consistency** (1), **Code Safety and Reliability** (9), **Change Method Access Level** (2), **Improve Type Safety** (1)
Support (New) Functionalities	Enhance (3) / Consolidate (2) **Functionality**, **Enhance Invocation Logic** (1), **Enhance Robustness** (3), **Support New Feature** (1), **Enhance Access Control** (1), **Introduce Necessary Structures for New Functionality** (1)
Other Specialized Goals	**Instance-specific Behavior Enhancement** (1), **Serialization Compatibility** (3), **Move Method** (1), **Collapse Hierarchy** (1), **Merge Attribute** (1), **Leverage Standard Library** (1), **Suppress Specific Compiler Warning** (3), **Implement Null Object Pattern** (1), Split Variable (1) / Package (1) / Attribute (1), **Update Attribute Value to Accommodate New Requirements or Changes** (1), **Add Modifier Correctly** (1), **Enable Interoperability** (1), **Update Annotation to Reflect a Different Attribute or Dependency** (1), **Centralize Common Functionality** (1)

E GLOSSARY WITH THE LIST OF ACRONYMS USED WITHIN THE STUDY

Table E.1. Glossary of the **global** acronyms used across the study, along with their definition.

Acronym	Concept definition	Acronym	Concept definition
AD	Anderson-Darling test	OSS	Open Source Software
BH	Benjamini-Hochberg test	PR	Pull Request
CA	Correlation Analysis	R	Recall (Only in Table 10)
CLI	Command-Line Interface	RAG	Retrieval-Augmented Generation
CoT	Chain of Thought	RF	Random Forest
DT	Dunn's all pairs test	RM	Refactoring Motivation
ELOC	Effective Lines of Code	RMC	Refactoring Motivation Category
FDR	False Discovery Rate	RMT	Refactoring Miner Tool
FI	Feature Importance	RQ	Research Question
FNR	False Negative Rate	RT	Refactoring Type
FWER	Family-Wise Error Rate	RW	Related Works
GT	Ground Truth	SE	Software Engineering
IG	Information Gain	SM	Software Metric
IRA	Inter-Rater Agreement	SW	Shapiro-Wilk test
LLM	Large Language Model	V1	First Validation Model
LRM	Large Reasoning Model	V2	Second Validation Model
MCTS	Monte Carlo Tree Search	V3	Third Validation Model
MDA	Mean Decrease in Accuracy	VCH	Version Control History
MDG	Mean Decrease in Gini	WT	Wilcoxon signed-rank test
MOSHO	Multi-Objective Spotted Hyena Optimizer	XGB	Extreme Gradient Boost
OOP	Object Oriented Programming		

F ANALYZING THE IMPACT OF ROOT-CANAL AND FLOSS REFACTORINGS

When it comes to refactoring motivation, an important goal underlying the refactoring activity relates to root-canal, i.e. refactorings related to quality-driven improvements, as well as floss refactorings, which consist on task-driven motivations to support other tasks, for instance, bug fixing commits. To address this, we revisited the entire collection of commits with detected refactoring activity which we obtained from mining the study context projects with the RefactoringMiner tool. Subsequently, we performed regular expression matching through the entire collection of commits with refactoring activity. Accordingly, 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" [28, 68]. Based on the structure of the tickets commonly used in well-known issue tracking systems such as GitHub or Jira, we used the following regular expressions:

- Issue ticket pattern for Jira: Matches issue tickets commonly used in Jira, where a project identifier is followed by the issue number:
  ```
  [A-Z]{2,}-\d+
  ```
- Issue ticket pattern for GitHub: Matches issue tickets commonly used in GitHub where the issue ticket number is preceded by a # symbol:
  ```
  #\d+
  ```

We acknowledge this approach to be the best effort, but it might not capture the totality of the bug-fixing commits given the existing heterogeneity when referencing issue tickets in commit messages. Building upon this newly mined data, we computed the distribution of refactoring commits detected as issue-fixing commits as they matched the defined patterns. We performed this process with the entire collection of refactoring commits, and did similarly with the commits used in the analyzed sample. Among all commits containing refactorings detected by RefactoringMiner, 54% were labelled as issue-fixing commits, while the remaining 46% could be considered as commits related with the introduction of new features. Results differed when we concentrated on the manually validated sample of 385 commits, with 28% of the analyzed commits being labelled as bug-fixing commits, thus leaving the remaining 72% of refactoring commits related to other tasks in spite of handling software bugs. These observational results suggest that our mined refactoring data, and therefore the spectrum of the motivations identified with our LLMs addresses both root-canal and floss goals of refactoring activity.

G ANALYZING THE EXTENT OF SELF-ADMITTED REFACTORING ACTIVITY WITHIN OUR MINED REFACTORING DATA

Prior research has demonstrated self-admitted refactoring to be a significant indicator of refactoring activity localization when investigating the context involving refactoring operations as well as the refactoring motivations declared within the natural language text stored in commit messages [4]. Consequently, we revisited the entire collection of commits with detected refactoring activity which we obtained from mining the study context projects with the RefactoringMiner tool. Subsequently, we performed regular expression matching through the entire collection of commits with refactoring activity. For that, we identified commits with self-admitted refactoring by employing regular expressions to check the presence of the keyword "`refactor*`". Building upon this newly mined data, we computed the distribution of refactoring commits detected as self-admitted refactoring commits as they matched the defined pattern. We performed this process with the entire collection of refactoring commits, and did similarly with the commits used in the analyzed sample. Among all commits containing refactorings detected by RefactoringMiner, only 4% of the commits with refactoring activity were labelled as self-admitted refactoring commits, while the remaining 96% of the commits did not show recognizable regular expression pattern. With respect to the analyzed sample of refactoring operations, the share of commits labelled as admitted refactoring commits ascended to a 12% of the observations, while the remaining 88% did not match with the defined regular expression. Although our regular-expression approach provides only an approximation based on the implemented best effort, the obtained results show that although it represents a minority of the mined and analyzed case, self-admitted refactoring operations exist in the analyzed data. Likewise, while we used developers' commit messages as context variant to feed the LLMs for identifying refactoring motivations, the present results suggest that most real-world refactoring operations require understanding further context rather than merely relying on commit messages, as these might not capture the entire goal of the performed commit.