UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AN ARTIFICIAL INTELLIGENCE–BASED APPROACH FOR COMPLETE EXTRACT METHOD RECOMMENDATIONS

GUISELLA CLARA ANGULO ARMIJO

ORIENTADOR: PROF. DR.VALTER VIERA DE CAMARGO

São Carlos – SP

October/2025

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AN ARTIFICIAL INTELLIGENCE–BASED APPROACH FOR COMPLETE EXTRACT METHOD RECOMMENDATIONS

GUISELLA CLARA ANGULO ARMIJO

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação, área de concentração: Engenharia de Software.

Orientador: Prof. Dr.Valter Viera De Camargo

São Carlos – SP

October/2025

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado da candidata Guisella Clara Angulo Armijo, realizada em 08/10/2025.

Comissão Julgadora:

Prof. Dr. Valter Vieira de Camargo (UFSCar)

Prof. Dr. Eduardo Magno Lages Figueiredo (UFMG)

Prof. Dr. Igor Scaliante Wiese (UTFPR)

Prof. Dr. Alan Demétrius Baria Valejo (UFSCar)

Prof. Dr. Daniel Lucrédio (UFSCar)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

I dedicate this work to my parents, who still teach me valuable life lessons

ACKNOWLEDGEMENTS

To God, for the opportunity to pursue this doctorate and for the strength to never give up.

To my advisor, Prof. Dr. Valter Vieira de Camargo, for the constant lessons throughout my doctoral journey, for his infinite empathy in the face of life's uncontrollable circumstances, and for his patience, encouragement, and guidance in developing high-quality scientific work.

To my family, especially my parents, Guillermo and Teófila, for their endless love, for the constant phone calls that warmed my heart on days of longing, and for their words of encouragement, care, and strength. You have always set the best example of perseverance, resilience, and courage. I am eternally grateful for shaping who I am, and I want to tell you that this achievement is also yours.

To my family in Brazil, and to my boyfriend Sandro, for his companionship, for his constant words of encouragement and his reassuring "tudo vai dar certo!", for always listening to me, for giving up many weekends so that I could finish my work, and for the immense support and love he shows me every day. To Mr. Wilson, Mrs. Joana, Angélica, Paula, Edilaine, Caio, and Rafael, thank you for your affection. I have no words to express my gratitude for your constant care and support.

To my soul sisters, Candy and Lizbeth, for the sincere friendship that began in our university classes back in Peru, when we dreamed of graduating and becoming successful professionals. Girls, we made it!

To all my colleagues from the "Advanse" research group, who always helped me clarify my doubts and never hesitated to offer their support. And to everyone who, in some way, contributed to the completion of this work, thank you!

Seja forte e corajoso! Não fique desanimado, nem tenha medo, porque eu, o Senhor, seu Deus, estarei com você em qualquer lugar para onde você for!

Josué 1:9

RESUMO

Sistemas de software devem evoluir continuamente para permanecer úteis, mas essa evolução frequentemente aumenta a complexidade e degrada a qualidade, tornando a refatoração essencial para a manutenção a longo prazo. Apesar das pesquisas em recomendações automáticas de refatoração, as abordagens existentes permanecem limitadas: em sua maioria tratam apenas de problemas específicos, como code smells, ou dependem de métricas de software, negligenciando as ricas representações semânticas e sintáticas do código-fonte. Além disso, geralmente oferecem apenas suporte parcial, como indicar onde refatorar sem especificar a ação a aplicar, ou sugerir um tipo de refatoração sem esclarecer sua justificativa. Esse escopo restrito ignora necessidades reais de refatoração e reduz a confiança dos desenvolvedores, comprometendo a utilidade prática. Este trabalho enfrenta essas limitações ao propor uma abordagem baseada em inteligência artificial para gerar recomendações completas de Extract Method. Fundamentada em refatorações históricas extraídas de projetos e representações semântica do código, a abordagem fornece recomendações que contemplam os critérios W3B, desenvolvidos nesta tese: Which (ação a aplicar), Where (onde aplicar), Why (por que é sugerida) e Benefits (benefícios obtidos). A metodologia seguiu um pipeline em múltiplas fases. Primeiro, foi construído de forma sistemática um conjunto especializado de exemplos de Extract Method a partir de repositórios de código aberto. Em seguida, desenvolveu-se um modelo de recomendação ajustando o CodeBERT para medir a afinidade entre fragmentos candidatos e o método analisado. Na terceira fase, foi projetado um explicador baseado em consenso, combinando SHAP, LIME e ANCHOR com um modelo substituto Random Forest para gerar explicações interpretáveis. Por fim, as saídas das fases foram integradas, concluindo a abordagem proposta e fornecendo recomendações completas e interpretáveis de Extract Method. A avaliação foi conduzida em um experimento controlado com pós-graduandos (7 participantes), que realizaram 24 avaliações ao analisar mais de um método cada. Os resultados mostraram melhora estatisticamente significativa nos escores de confiança ($p = 0{,}011$), forte alinhamento entre as sugestões da ferramenta e as escolhas dos participantes (57,7%) e alta concordância (96%) quanto aos benefícios apontados, incluindo legibilidade, modularidade e manutenibilidade.

Keywords: recomendação de refatorações, inteligência artificial, explicável IA

ABSTRACT

Software systems must continually evolve to remain useful, but this evolution often increases complexity and degrades quality, making refactoring essential for long-term maintainability. Despite significant research on automated refactoring recommendations, existing approaches remain limited: the vast majority focus on solving specific problems like code smells or rely on software metrics, while neglecting the rich semantic and syntactic representations of source code. Moreover, they usually provide partial support, such as identifying where to refactor without specifying what refactoring to apply or suggesting a refactoring type without clarifying its rationale. This narrow scope not only overlooks diverse and real refactoring needs but also undermines developer trust, reducing the practical usefulness of such tools. This work addresses these limitations by proposing an artificial intelligence–based approach for generating complete recommendations for Extract Method refactoring. Grounded in real refactorings applied in the past of the projects and leveraging the semantic representation of code, the approach delivers recommendations that explicitly cover the W3B criteria, developed in this work: Which refactoring to apply, Where in the code it should be applied, Why it is suggested, and the Benefits it brings. The methodology followed a multi-phase pipeline. First, a specialized dataset of Extract Method samples was systematically built. Second, a recommendation model was developed by fine-tuning CodeBERT to measure the affinity between candidate fragments and the analyzed method. Third, a consensus-based explainer was designed, aggregating outputs from SHAP, LIME, and ANCHOR with a Random Forest surrogate to provide interpretable explanations. Finally, the outputs of all phases were integrated into a concluding phase, completing the proposed approach and delivering complete and interpretable Extract Method recommendations. Evaluation was conducted through a controlled experiment with postgraduate students (7 participants out of 9 initially enrolled), who provided a total of 24 evaluations by assessing more than one method each. Results showed a statistically significant improvement in participants' confidence scores ($p = 0.011$), strong alignment between tool suggestions and participants' own choices (57.7%), and high agreement (96%) with the stated benefits, including readability, modularity, and maintainability.

Keywords: refactoring recommendations, artificial intelligence, Explainable AI (XAI)

List of Figures

List of Tables

LIST OF LISTINGS

*

TABLE OF CONTENTS

Chapter 1
INTRODUCTION

1.1 Context

As software systems evolve, they inevitably accumulate complexity and degrade in quality (MENS; TOURWÉ, 2004). According to Lehman's first law, systems must be continually adapted or they become progressively less satisfactory (LEHMAN, 1980). To address this, refactoring has long been employed as a disciplined practice to counteract design erosion and support maintainability.

Refactoring, popularized by Martin Fowler in 1999 in the book *Refactoring: Improving the Design of Existing Code* (FOWLER et al., 1999) is defined as the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Refactoring prevents design degradation and reduces the effort of fixing errors. It also accelerates development by maintaining code health, improving readability and making the system easier to understand and maintain (FOWLER, 2018). Despite its benefits, identifying which structures to refactor remains a challenging and time-consuming task, particularly in large or complex projects. The most common technique studied in the literature for identifying refactoring opportunities is code smells, which are defined as the surface indications within software code that suggest a deeper problem exists in the system (FOWLER, 2018).

Although the identification of refactoring opportunities generally depends on the software developer's skills and insights, this process may still be supported by refactoring recommenders. These recommendations can facilitate the process of detecting

the components (classes, methods, variables, etc.) that need refactoring.

Research on refactoring recommendations dates back to the early 2000s. Early approaches in literature rely on heuristics-based methods that uses practical rules and structural or behavioral patterns in the source code (e.g., dependencies between methods and classes, method responsibilities, or code organization) to estimate software quality properties and identify refactoring opportunities (FOKAEFS et al., 2007) (TERRA et al., 2018). Search-based techniques employ variety of evolutionary and genetic algorithms that search the code space to maximize an optimization function. Researchers have proposed using Pareto optimization, which simplifies metric integration and provides users with multiple optimal refactoring options (NYAMAWE et al., 2020)(ALIZADEH et al., 2019) (ALIZADEH; KESSENTINI, 2018).

More recent approaches have leveraged Machine Learning methods in the realm of software engineering to present new avenues for the automation of the identification of refactoring opportunities. Researchers have employed these techniques that utilize source code metrics to train algorithms for different types of refactorings. Supervised learning techniques are the most commonly used, since the algorithms learn how to identify the opportunities for refactoring based on metrics and fixed thresholds. These metrics focus predominantly on structural elements like LOC (Lines of Code) , number of assignments, number of iterators, among others (KURBATOVA et al., 2020) (YUE et al., 2018) (LIU et al., 2018) (IMAZATO et al., 2017) (KUMAR; SUREKA, 2017) (PANIGRAHI et al., 2020).

Despite these advances, important challenges remain. Most approaches still focus primarily on detecting code smells (CUI et al., 2023) (ALOMAR et al., 2023) (SAHEB-NASSAGH et al., 2022) (SHENEAMER, 2020) (LIU et al., 2018) (XU et al., 2017a). However, code smells address only part of the refactoring spectrum and neglect other crucial objectives, such as reusability, testability, and readability (HENRIQUE et al., 2023). Furthermore, existing recommendations are frequently incomplete. They typically fail to specify which refactoring should be applied, for example, Move Method and Extract Method; where it should be applied, that is, the source code involved in the refactoring, for example, the identification of the methods, classes, variables, the identification of the code fragments or the variables; and why the recommendation is suggested, meaning the rationale behind the decision and the benefits of applying the

recommendation.

We argue that a high-quality recommendation is one that provides developers with all the relevant information they need to make a decision. Therefore, in this work, we employ the term **complete** to refer to recommendations that not only specify the name of the refactoring to be applied but also identify which software elements are involved in the refactoring, explain the rationale behind the recommendation, and outline the potential benefits. In this regard, Alizadeh et al. found that although existing tools can detect hundreds of code-level issues (e.g., anti-patterns and low-quality attributes), they often fail to clearly indicate where to start and how the suggested refactorings are relevant in the given context (ALIZADEH et al., 2019).

These challenges highlight the need for more advanced recommendation approaches. Recent advances in neural networks, particularly Large Language Models (LLMs), offer promising alternatives to overcome such challenges. LLMs are deep neural networks designed to understand, generate, and manipulate natural language at scale. Using architectures such as Transformers, LLMs learn complex linguistic patterns and contextual dependencies from massive corpora. A special type is the re-trained Language Models (PLMs) which are models that undergo large-scale pre-training on general tasks, such as Masked Language Modeling (MLM) or Next Token Prediction, before being fine-tuned for specific downstream applications (GOODFELLOW et al., 2016) (NYIRONGO et al., 2024).

In particular, Pre-trained Language Models (PLMs) such as CodeBERT (FENG et al., 2020) have gained attention for their ability to learn rich semantic and syntactic representations of source code. Unlike traditional metric- or heuristic-based techniques, PLMs can capture deeper contextual information, enabling recommendations that are more interpretable and aligned with developers' expectations. This opens the possibility of generating Extract Method recommendations that go beyond surface-level indicators, providing support that is both technically sound and more trustworthy to practitioners.

This thesis leverages these advances to propose an AI-based approach for generating complete and interpretable Extract Method recommendations. A key contribution is the W3B criteria, developed in this thesis as a product of the analysis of the systematic review conducted, to define what constitutes a complete recommendation: Which refac-

toring to apply, Where in the code, Why it is suggested, and the Benefits it brings. The proposed approach is structured into four phases: (i) building the Extract Method dataset; (ii) training the pre-trained CodeBERT model; (iii) designing a consensual explainer to generate explanations for the recommendation; and (iv) ensembling the outputs of the phases to produce complete and interpretable recommendations that address the real needs of developers.

1.2 Motivations for Developing this Thesis

The main motivation behind this thesis lies in the **incomplete recommendations for *Extract Method* refactoring**. In this context, the term *incomplete* refers to recommendations that address only part of the refactoring elements. For instance, they may identify the method to be refactored but omit the fragment to extract or indicate the method without specifying the refactoring to be applied. Such gaps lead developers to question the quality of the recommendations, often resulting in their rejection due to the absence of sufficient information. Although several initiatives have been proposed in the literature, they still present important limitations, as identified in the Systematic Review presented in Chapter 3. This limitation reduces both confidence in and the usefulness of automated recommendation tools, as developers may not fully understand the recommendations and must still determine the missing elements themselves. To address this issue, this thesis is guided by three specific motivations, each addressing a different aspect of the limitation.

1. **Recommendation for a subset of problems**. We realized in literature that the focus of most refactoring recommendation approaches are based on solving code Smells (CUI et al., 2023) (ALOMAR et al., 2023) (NYAMAWE, 2022) (SAHEBNASSAGH et al., 2022) (KURBATOVA et al., 2020) (PANIGRAHI et al., 2020) (SHENEAMER, 2020) (YUE et al., 2018) (LIU et al., 2018) (XU et al., 2017a). In such cases, the model learns how to identify the smell and suggests a refactoring to solve it. The recommended refactoring corresponds to the classic way of solving a code smell. For example, for solving a Feature Envy smell the developer should apply the Move Method refactoring. While useful, this perspective may overlook real and diverse problems that extend beyond predefined smells. We argue that analyzing refactorings applied in real development histories can uncover more

representative problems and provide recommendations that are more relevant and aligned with developers' practical needs. Thus, beyond addressing traditional smells, we aim to capture a broader spectrum of real refactoring motivations.

2. **The need for better formulation of the recommendation.** The usefulness of a refactoring recommendation depends not only on its accuracy but also on how it is presented to developers. Existing approaches vary considerably in their formulation. Some specify only *which* refactoring should be applied (NYAMAWE, 2022) (NYAMAWE et al., 2020) (SIDHU et al., 2022), providing only the name of the refactoring. Others indicate only *where* the refactoring should be applied (PANIGRAHI et al., 2022) (ALENEZI et al., 2020) (KUMAR et al., 2015) (KUMAR; SUREKA, 2017), identifying the components involved in the refactoring. While many neglect to explain *why* the refactoring is suggested, meaning the rationale behind the recommendation, or directly present the *benefits* it may provide. A truly effective recommendation should integrate all these criteria. Our systematic review enabled the definition of the **W3B** criteria (Which, Where, Why, and Benefits) as the foundation of a complete recommendation. This motivates the development of approaches that explicitly incorporate all **W3B** criteria, ensuring that recommendations are actionable and trustworthy.

3. **The need for better comprehension of the reasons behind the recommendations**. This motivation is articulated to the **Why** of the **W3B** criteria. Thus, while machine learning models have shown remarkable performance in different software engineering tasks, many of them function as *black boxes*, offering little to no insight into their internal decision-making processes (HASSIJA et al., 2024). This lack of transparency is problematic for the refactoring recommendation approaches since developers need to understand and trust the rationale behind the suggested refactoring. Explainable Artificial Intelligence (XAI) seeks to bridge this gap by making model decisions more transparent and interpretable (ROY et al., 2022a; BOMMER et al., 2024; GUNNING et al., 2019). Techniques such as Local Interpretable Model-Agnostic Explanations (LIME) (RIBEIRO et al., 2016), SHapley Additive explanations (SHAP) (LUNDBERG; LEE, 2017b), and Anchors (RIBEIRO et al., 2018) can be employed to clarify why a given refactoring is recommended. Beyond delivering accurate suggestions, effective tools must foster

developer confidence and acceptance, which can only be achieved by coupling recommendations with clear and trustworthy explanations.

In summary, the motivations behind this thesis are rooted in the need for more complete and understandable refactoring recommendations. By going beyond traditional code smells, incorporating explainability, and ensuring a comprehensive formulation, this work seeks to bridge the gap between current automated approaches and the actual needs of software developers.

1.3 Goals

Our general goal is to increase developers' trust and acceptance of refactoring recommendations by providing complete and understandable recommendations for Extract Method. This involves delivering recommendations that not only indicate what refactoring to apply and where, but also explain why it is suggested and the benefits it brings, ensuring that developers can make informed decisions with confidence. To achieve this, we define the following five specific goals:

GOAL 1: Deliver an approach for Extract Method refactoring recommendations that generates complete and comprehensive recommendations that conform to the **W3B** criteria, based on real refactorings previously applied in projects and leveraging the semantic representation of the code.

GOAL 2: Deliver a strategy to generate interpretable explanations integrated into the Extract Method recommendation approach, leveraging Explainable AI techniques to enhance developer trust, confidence, and adoption.

GOAL 3: Deliver criteria to measure a complete refactoring recommendations. Based on the analysis of our systematic review, we established the **W3B** criteria, specifying **which** action to apply, **where** in the code, it should be applied, **why** it is suggested, and the benefits it brings. These criteria provide the foundation for designing and evaluating complete recommendations.

GOAL 4: Characterize the state of the art in machine learning–based refactoring recommendation approaches, through a systematic review highlighting current strengths, limitations, and open challenges.

1.4 Contribution

Theoretical Contribution The key contribution lies in enabling recommendations that developers can both trust and understand, thereby fostering confidence not only in the suggested refactorings but also in the tool that provides them. More specifically, the thesis contributes the following:

- **An approach for generating Extract Method recommendations** that formulates complete recommendations covering **which**, **where**, **why**, and **benefits**, based on evidence from past real-world refactorings;
- **A strategy to address the disagreement problem in explainability models**, improving the consistency and reliability of explanations provided to developers;
- **A criteria** for measuring the completeness of the recommendation. The criteria is called (W3B).
- **A systematic review of refactoring recommendation approaches**, conducted following the guidelines proposed by Keele University staff (KEELE et al., 2007), which identifies current gaps, limitations, and opportunities for advancing the field.

Technical contribution:

- A refactoring recommendation tool, implemented as an API, that provides complete recommendations for Extract Method refactorings while offering a user interface for interaction;
- A dataset of real Extract Method refactorings, systematically collected from open-source software repositories, which can be reused for future research and benchmarking;

- An experimental evaluation with postgraduate students, designed to assess how explanatory support influences developers' trust, confidence, and acceptance of refactoring recommendations;
- A modular pipeline combining FME (JAR) for code extraction, CodeBERT for learning from refactorings, and Explainable AI techniques for explanation, which can serve as a foundation for future extensions in recommendation systems.

1.5 Publications

- ARMIJO, G.; SANTIBAñEZ, D.; DURELLI, R.; CAMARGO, V. On the employment of machine learning for recommending refactorings: A systematic literature review. In: Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software. Porto Alegre, RS, Brasil: SBC, 2024. ISSN 0000-0000.
- ARMIJO, G. A.; CAMARGO, V. V. de. Refactoring recommendations with machine learning. In: SBC. Simpósio Brasileiro de Qualidade de Software (SBQS). [S.l.], 2022. p. 15–22.

1.6 Structure

The remainder of this thesis is structured as follows:

- Chapter 2 provides the necessary background to understand the concepts and techniques used in the subsequent chapters;
- Chapter 3 presents a systematic review and mapping of the use of Machine Learning for recommending refactorings. This chapter provides an overview of existing approaches and the evidence that motivates this work;
- Chapter 4 introduces the W3B criteria, defining the essential elements of a complete refactoring recommendation and establishing a baseline for assessing recommendations;

- Chapter 5 presents the Consensual Explainer, a strategy to generate consistent and interpretable explanations for the recommendations;
- Chapter 6 describes the proposed refactoring recommendation approach in detail, including the methodology used to build it;
- Chapter 7 presents the implementation of the Refactoring Recommendation API in detail, including its interface, architecture, and components;
- Chapter 8 presents the controlled experiment with practitioners for evaluating the refactoring recommendation tool;
- Chapter 9 summarizes the findings of this thesis, discusses its contributions, and provides directions for future work.

Chapter 2
BACKGROUND

2.1 Initial considerations

In this chapter, we summarize the state-of-the-art in our chosen area of research. This chapter is divided into four parts: first, in Section 2.2 we introduce concepts related to Software Refactoring and Code Smells. Second, in Section 2.3 we provide a clear definition of concepts related to Machine Learning. Third, in Section 2.4 we provide the concepts related to the pre-trained CodeBERT model, and Section 2.5 introduces the foundation of Explainable Machine Learning (XAI).

2.2 Software Refactoring and Code Smells

The term refactoring was originally introduced in 1992 by Opdyke in his PhD thesis (OPDYKE, 1992). However, this "term"gained popularity when Fowler published his famous book Refactoring - Improving the Design of Existing Code (FOWLER et al., 1999) where he defined refactoring as “the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure” (FOLWER, 1999).

In literature, the techniques and tools for software refactoring are classified into three main categories: manual, semi-automated and fully-automated approaches. In manual refactoring, the developers refactor with no tool support, identifying the parts of the program that require attention and performing all aspects of the refactoring by

hand. In fully-automated refactoring, developers provide their code as input, and the tool will provide refactoring recommendations automatically (MENS; TOURWÉ, 2004) (ALIZADEH et al., 2019). Finally, in semi-automated refactoring, the tool gives some support for the refactoring process.

The manual refactoring process becomes a time consuming activity since developers have to analyze the source code to (SILVA et al.,) (MENS; TOURWÉ, 2004): (i) Identify where the software should be refactored; (ii) Determine which refactoring(s) should be applied to the identified places. As an example, splitting a non cohesive class into different classes with strongly related responsibilities (i.e., Extract Class refactoring) requires the analysis of all the methods of the original class to identify groups of methods implementing similar responsibilities, which should be grouped together in the new classes to be extracted. This task becomes harder when the size of the class to split increases and its cohesion decreases (BAVOTA et al., 2014); (iii) Guarantee that the applied refactoring preserves behavior; (iv) Apply the refactoring; and (v) Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability).

The hardest task in the refactoring process is to identify what part of the code must be refactored. Thus, code smells have been used as indicators to locate the piece of code that probably needs the application of refactorings. According to Fowler, a code smell is a surface indication that usually corresponds to a deeper problem in the system (FOWLER, 2018). Smells can be introduced in software systems for many circumstances, including developers' lack of skill or awareness, frequently changing requirements, priority to features over quality, among others (SHARMA; SPINELLIS, 2018).

Although code smells have been used as indicators to know where to refactor, the reality is that the refactorings have been applied to solve more problems than code smells. For instance, for the specific case of the Extract Method refactoring, Henrique et al. (HENRIQUE et al., 2021) mined the extract method refactoring and analyzed commit messages to capture the motivations behind the refactoring. Remember that the extract method aims at extracting a snippet or fragment of source code from a given method and taking it to a new method. So, the authors identified 11 motivations behind the application of the Extract Method: (1) remove duplication, (2) code cleanup, (3) organize code, (4) improve readability, (5) improve testability, (6) reduce the method, (7) feature update, (8)

add feature, (9) bug fix, (10) improve performance, and (11) removal of features.

2.3 Machine Learning

Machine Learning (ML) uses computers to simulate human learning and allows computers to learn from the real world and improve the performance of some tasks based on this new knowledge. More formally, ML is defined as follows: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"* (MICHALSKI et al., 1985).

There are two main types of learning processes: supervised and unsupervised. In supervised learning, algorithms are provided with both training data and the corresponding correct answers, allowing the model to learn from examples. This paradigm is commonly subdivided into two categories: *Classification* and *Regression*.

In the case of *Classification*, the task of the machine learning algorithm is to infer patterns from labeled training data and then apply the acquired knowledge to unseen real-world data (KOTSIANTIS et al., 2007). In formal terms, the objective is to learn a mapping from inputs x to outputs y, where $y \in \{1, \ldots, C\}$, with C denoting the number of possible classes.

If $C = 2$, this is called *binary classification* (in which case we often assume $y \in \{0, 1\}$). An illustrative example is spam detection in emails. In this task, the system receives an email as input (represented by features such as the frequency of certain words, sender information, or presence of links) and must decide whether the email is "spam" or "not spam". Here, the output is limited to two possibilities: $y = 0$ for "not spam" and $y = 1$ for "spam". This type of problem is common in many real-world scenarios, such as medical diagnosis with two outcomes (disease vs. no disease) or sentiment analysis with two classes (positive vs. negative).

If $C > 2$, this is called *multiclass classification*. A well-known example is handwritten digit recognition, such as in the MNIST dataset. In this case, the input is an image of a handwritten digit, and the goal is to assign it to one of ten possible classes, corresponding to the digits 0 through 9. Thus, the output space is larger, $y \in$

$\{0, 1, 2, \ldots, 9\}$, and the model must learn to discriminate among multiple categories rather than just separating two groups (MURPHY, 2012).

On the other hand, *Regression* is similar to classification, except that the response variable is continuous rather than categorical. Formally, the goal is to learn a function $f : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X}$ denotes the input space and the output is a real-valued variable. For example, one may attempt to predict tomorrow's stock market price given current market conditions and other side information.

In unsupervised learning, machine learning algorithms are not provided with a training set containing input–output pairs. Instead, they are presented with raw data about the real world and must learn structure from it autonomously. Such algorithms are mainly focused on identifying hidden patterns, regularities, or relationships within the data. For example, suppose that an algorithm has access to user profile information in a social network. By applying unsupervised learning, it can separate users into personality categories, such as outgoing and reserved, thereby allowing the company to design more effective strategies, such as targeted advertising (CELEBI; AYDIN, 2016).

A central task in unsupervised learning is *cluster analysis*, also called *data segmentation*. The main idea is to group or partition a collection of objects into subsets, or "clusters," in such a way that objects within the same cluster are more closely related to each other than to those in different clusters. An object can be characterized either by a set of measurements (features) or by its similarity relations to other objects. Formally, the goal of cluster analysis is to partition observations so that pairwise dissimilarities within the same cluster are minimized, while dissimilarities across clusters are maximized (HASTIE et al., 2009).

Clustering algorithms can be broadly categorized into three groups: *combinatorial algorithms*, *mixture modeling*, and *mode seeking*. Among these, one of the most widely used techniques is the *K-means algorithm*. K-means is an iterative descent method designed for quantitative variables, in which objects are grouped by minimizing the squared Euclidean distance between each object and the centroid of its assigned cluster. Despite its simplicity, K-means has become a cornerstone technique for practical applications of unsupervised learning (HASTIE et al., 2009).

On the other hand, Artificial Neural Networks (ANN) are a subfield of ML

that have been motivated from their inception by the recognition that the human brain computes in entirely different ways from conventional digital computers. So, an ANN can be defined as: A massively parallel combination of simple processing units that can acquire knowledge from the environment through a learning process and store the knowledge in its connections (SYMON, 1999). In the same way, Deep Learning (DL) is a subfield of ANN and is essentially a statistical technique for classifying patterns based on sample data, using neural networks with multiple layers.

2.4 Pre-trained CodeBERT Model

Large Language Models (LLMs) are deep neural networks designed to understand, generate, and manipulate natural language at scale. Using architectures such as Transformers (VASWANI et al., 2017), LLMs learn complex linguistic patterns and contextual dependencies from massive corpora. Despite their versatility in tasks like text generation, summarization, question answering, and code-related applications, LLMs require large amounts of labeled data and substantial computational resources for task-specific training. Examples include GPT-3, GPT-4, and Codex, which illustrate both the capabilities and practical limitations of large-scale models.

Pre-trained Language Models (PLMs) are models that undergo large-scale pre-training on general tasks, such as Masked Language Modeling (MLM) or Next Token Prediction, before being fine-tuned for specific downstream applications. This strategy reduces the need for extensive labeled datasets and allows for rapid adaptation to specialized domains. Examples include BERT for natural language (DEVLIN et al., 2019) and CodeBERT or Codex for programming languages (SVYATKOVSKIY et al., 2021; FENG et al., 2020), which integrate source code and natural language documentation to learn rich semantic representations. In software engineering, PLMs for code can be considered LLMs for programming, combining large-scale learning, pre-training, and contextual understanding of code.

The Transformer architecture, introduced by Vaswani et al. (VASWANI et al., 2017), is the backbone of these models. Unlike traditional recurrent models, Transformers use attention mechanisms to capture long-range dependencies in sequences, allowing tokens to be processed efficiently in parallel. This mechanism is particularly suitable

for source code, which often exhibits complex hierarchical structures and non-linear dependencies among variables, functions, and control blocks. By enabling each token to attend to all others in the sequence and weight their relative importance, Transformers effectively capture both syntactic and semantic relationships.

Building on this foundation, CodeBERT was developed as a pre-trained model specialized for programming languages. It jointly models source code and natural language documentation (e.g., comments) to generate unified semantic representations that reflect both syntactic structure and functional meaning. CodeBERT is pre-trained using masked language modeling on large-scale code corpora spanning multiple programming languages, allowing it to generalize across tasks such as code search, code completion, code summarization, and function classification (FENG et al., 2020). Its ability to integrate natural language and code semantics makes it particularly effective for software engineering applications, including recommendation systems and automated refactoring tools.

In practical software engineering, CodeBERT has been successfully applied to a variety of tasks. For instance, it can assist developers in code completion by suggesting syntactically correct and semantically coherent snippets based on partially written code (SVYATKOVSKIY et al., 2021). In code search, CodeBERT enables developers to retrieve relevant code fragments by querying with natural language descriptions, bridging the gap between human understanding and code representation (FENG et al., 2020). The model has also been used for automated code summarization, generating concise natural language descriptions of functions and methods, which facilitates documentation and code comprehension (FENG et al., 2020). Additionally, CodeBERT can support function classification and clone detection by identifying semantically similar code fragments across large codebases, which is valuable for refactoring and maintenance (FENG et al., 2020).

2.5 Explainable Machine Learning

With the rise of developments in Artificial intelligence (AI) and Machine Learning (ML) algorithms, researchers from various application domains have shown increasing interest in taking advantage of these algorithms. As a result, AI and ML are

being used today in many application domains (ISLAM et al., 2022). However, there is an inherent conflict between ML performance (e.g., accuracy) and explainability. Often, the highest performing methods (e.g., Deep Learning) are the least explainable, while the most explainable (e.g., decision trees) are the least accurate (GUNNING et al., 2019). As a consequence, most of the existing ML algorithms can often lead to robust and accurate models from data, but in application terms, they fail to provide end-users with descriptions of how they were built or to produce convincing explanations for their predictions (LONGO et al., 2020).

Since 2017, explainable AI (XAI) has been gaining more attention due to the widespread application of ML. XAI is a research field that aims to make ML systems's results more understandable to humans. According to DARPA (GUNNING, 2017), XAI aims to produce more explainable models while maintaining a high level of learning performance (prediction accuracy); and enabling human users to understand, appropriately trust, and effectively manage the emerging generation of artificially intelligent partners. FICO (FICO, 2018) defined XAI as an innovation aimed at opening up the black box of ML and a challenge to create models and techniques that are both accurate and provide trustworthy explanations that satisfy customers' needs.

In this context, one of the main reasons for producing an explanation is to gain the trust of the users, create confidence in the system, and make them feel comfortable while controlling and using it (LONGO et al., 2020). So, XAI should be able to answer 'why a particular output was obtained?', providing trustworthiness, transparency, confidence, and informativeness (GOHEL et al., 2021).

XAI is not an isolated field; it is the intersection of three major research domains: Machine Learning (ML), Human-Computer (HCI) and Social Science. Thus, a system that provides a causal explanation of its inferential process is perceived as more human-like by end-users due to the innate tendency of human psychology. With this purpose, Phillips et al. (PHILLIPS et al., 2020) define four principles to create effective, more human-understandable explainable AI systems: (i) Explanation. Systems deliver accompanying evidence or reasons for all outputs; (ii) Meaningful. Systems provide explanations that are understandable to individual users; (iii) Explanation Accuracy. The explanation correctly reflects the system's process for generating the output; (iv) Knowledge Limits. The system only operates under conditions for which it was designed

or when it reaches a sufficient confidence in its output.

Regarding taxonomy, there are several classifications in the literature. Figure 1 shows a high-level XAI ontology proposed by Angelov et al. (ANGELOV et al., 2021). The figure introduces a variety of concepts that converge in explainable models. With the term Transparent Models, the authors refer to models where the decisions are often transparent, although transparency, as a property, is not sufficient to guaranty that a model will be readily explainable. Typical transparent models include k-nearest neighbors (kNN), decision trees, rule-based learning, Bayesian networks, and so on.

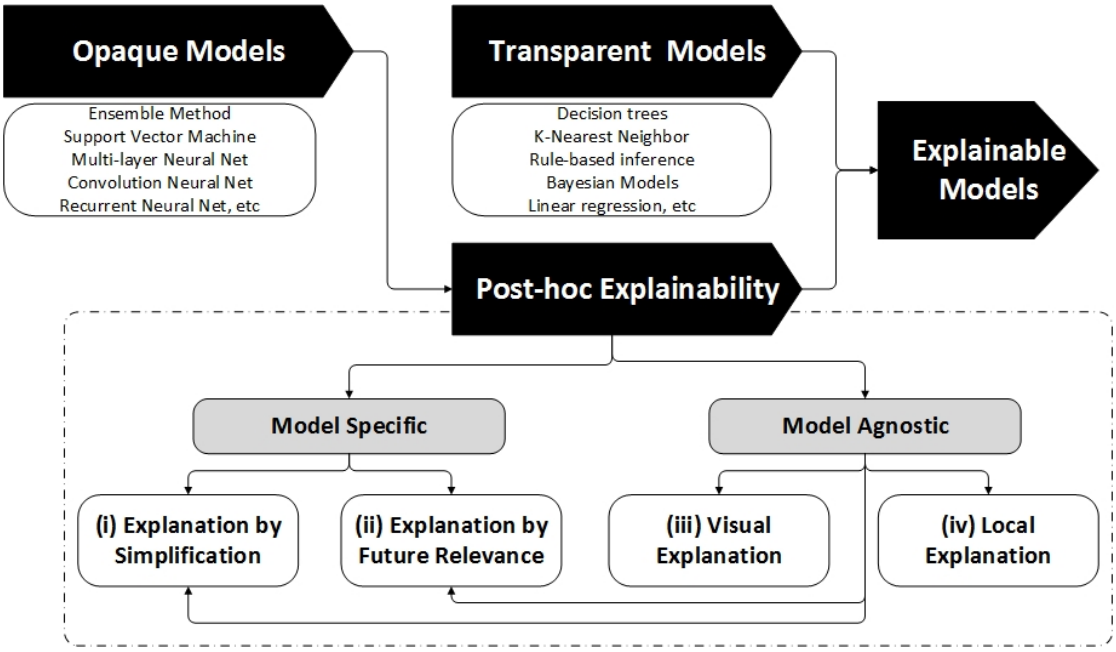


Figure 1 – The high-level ontology of explainable artificial intelligence approaches adapted from (ANGELOV et al., 2021)

We explain the sub-types of models as follows:

(i) Explanation by simplification: By simplifying a model via approximation, we can find alternatives to the original models to explain the prediction we are interested in. For example, we can build a linear model or a decision tree around the predictions of a model, using the resulting model as a surrogate to explain the more complex one; (ii) Explanation by feature relevance: This idea is similar to simplification. Roughly, this

type of XAI approaches attempts to evaluate a feature based on its average expected marginal contribution to the model's decision, after all possible combinations have been considered.

(iii) Visual explanation: This type of XAI approach is based on visualization. As such, the family of data visualization approaches can be exploited to interpret the prediction or decision over the input data; and (iv) Local explanation: Local explanations approximate the model in a narrow area, around a specific instance of interest, and offer information about how the model operates when encountering inputs that are similar to the one we are interested in explaining.

Using the taxonomy proposed by Angelov et al. the most popular explainability methods are classified as follows.

LIME (Local Interpretable Model-Agnostic Explanations) is one of the most popular interpretability techniques classified as Local Explanation. It is an algorithm that can explain the predictions of any classifier or regressor in a faithful way by approximating it locally with an interpretable model. It modifies a single data sample by tweaking the feature values and observes the resulting impact on the output. It performs the role of an "explainer"to elucidate predictions from each data sample. The output of LIME is a set of explanations representing the contribution of each feature to a prediction for a single sample, which is a form of local interpretability (DIEBER; KIRRANE, 2020). Figure 2 - *A* shows the model predicting 17% probability for *No Diabetes* and 83% for *Diabetes*. Figure 2 - *B*, the bar chart highlights each variable's contribution: glucose (0.61), BMI (0.42), diabetes pedigree function (0.20), and blood pressure (0.37) were the main factors increasing the probability of diabetes. Insulin, skin thickness, number of pregnancies, and age had less impact. Figure 2 - *C*, a table presents the actual values of the features used in the explanation. This example illustrates how LIME provides local interpretability, showing why the model favored the Diabetes classification in this specific case.

SHAP (SHapley Additive exPlanation) is classified as an explanation based on feature relevance. It is a game-theoretic approach to explaining ML predictions. SHAP seeks to deduce the extent to which each feature contributed to a decision by representing the features as players in a coalition game. The payoff of the game is

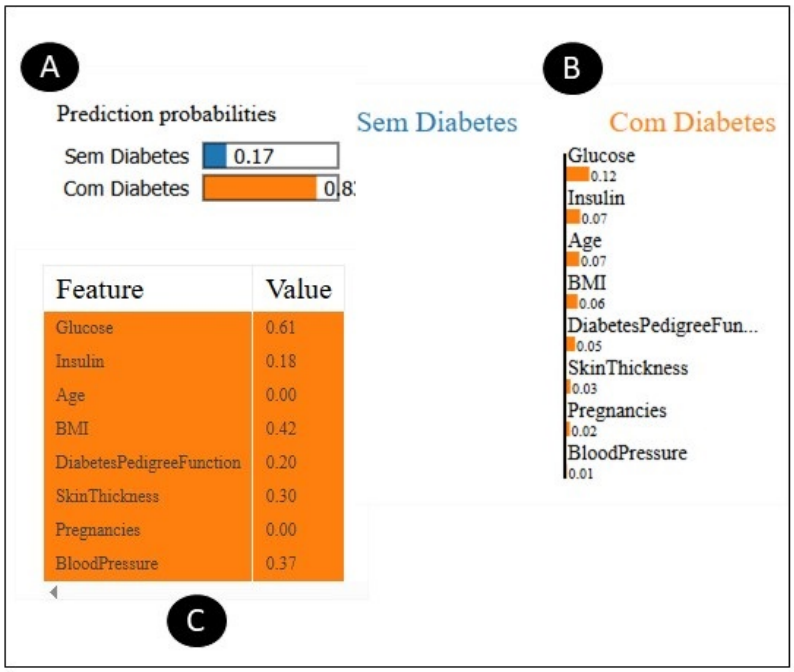


Figure 2 – Example of LIME explanation

an additive measure of importance, the so-called Shapley value, which represents the weighted average contribution of a particular feature within every possible combination of features (LUNDBERG; LEE, 2017b).

The ANCHOR methods explain any black-box classification model's predictions by identifying decision boundaries that sufficiently "anchor" the prediction. A rule that effectively ties a prediction to a specific local context within the examined instance, known as an anchor explanation, ensures that modifications to other feature values do not significantly impact the rule's ability to explain the prediction. The Anchors technique relies on reinforcement learning methods and a graph search algorithm to minimize the number of model calls required during runtime and efficiently recover from local optima (RIBEIRO et al., 2018).

Chapter 3

SYSTEMATIC REVIEW AND RELATED WORK

3.1 Initial Considerations

In this section, we present the systematic review and the related works relevant to this thesis. The systematic review provides a comprehensive overview of existing approaches in the literature, enabling the identification of trends, strengths, and gaps in current research. To guide the review, we defined three main research questions along with five sub-questions aimed at exploring how Machine Learning techniques have been applied in the context of refactoring recommendations.

3.2 Systematic Review

This systematic review aims to analyze how Machine Learning (ML) algorithms have been applied in the context of Refactoring Recommendations. To ensure methodological rigor and transparency, the review was conducted following the well-established guidelines proposed by the Keele University staff (KEELE et al., 2007), which structure the process into three distinct phases: planning, conducting, and reporting the review.

In the planning phase, the objectives of the study were clarified, and specific research questions were formulated to guide the review. The conducting phase involved executing comprehensive search strategies across multiple digital libraries and databases, applying clearly defined inclusion and exclusion criteria to select relevant primary studies. During this phase, studies were screened, selected, and subjected to quality assessment.

Subsequently, data extraction was performed to collect key information from each study, followed by the synthesis and analysis of the extracted data.

Lastly, the reporting phase focused on organizing and presenting the results in a clear and structured manner. It also defined the strategy for disseminating the findings, aiming to reach both academic and practitioner communities. This included the preparation of detailed documentation, visual summaries, and discussions to highlight trends, gaps, and future research directions in the use of Machine Learning for refactoring recommendations.

3.2.1 Planning

3.2.1.1 Research questions

We defined three main research questions, each accompanied by relevant sub-questions, with the goal of investigating key aspects related to the use of Machine Learning techniques in the context of refactoring recommendations. The research questions are as follows:

RQ1 *Which relationships between Machine Learning and Refactorings Recommendations are explicitly discussed in the literature?*

- **RQ1.1** *Which refactorings have been investigated?* This question allows us to identify which types of refactorings have received the most attention and to analyze their characteristics, helping us to infer possible reasons for the focus on certain refactorings over others.
- **RQ1.2** *Which ML algorithms have been used to recommend refactorings?* The aim here is to identify the Machine Learning techniques adopted and explore possible connections between these techniques and the effectiveness or scope of the recommendations.
- **RQ1.3** *How datasets and features are classified?* This question provides insights into two dimensions: (i) the variety of repositories and systems used to construct the datasets, and (ii) the types of features extracted, such as traditional code metrics, historical data, and syntactic or semantic characteristics.

RQ2 *Do the way the approaches are evaluated depend on the automation level of them?*

- **RQ2.1** *How have the approaches been evaluated?* This question aims to assess the maturity level of the tools utilized in the approaches.
- **RQ2.2** *What is the automation level (fully automated or semi-automated) of the approaches?* Answering this question provides insights into the evaluation methods employed, offering valuable clues about the most common techniques used to assess the refactoring recommendations.

RQ3 *Have the approaches concerned with the quality of the recommendations?* Answering this question provides insights into how researchers are taking advantage of ML to develop a recommendation engine, optimizing the resulting recommendations.

3.2.1.2 Search string

Figure 3 shows the base string elaborated around three terms: (*i*) Recommendation; (*ii*) Refactoring, and (*iii*) Machine Learning. This base string was adapted considering alternative spellings and synonyms for each of the 5 digital libraries: Scopus, IEEE Xplore, ACM, Science Direct and Wiley. Therefore, we restricted the search to the period between 2015 and 2023 and, to avoid missing important/relevant articles, we conducted a backward snowball technique using reference lists from the final set of articles.

Figure 3 – Search string

```
  ("Recommendation" OR "Recommend" OR "Recommending" OR
    "Identification" OR "Identify" OR "Identifying" OR
 "Prediction" OR "Predict" OR "Predicting" OR "Prevision")
AND ( "Refactoring" OR "Refactor") AND ( "Machine Learning"
    OR "Supervised Learning" OR "Unsupervised Learning")
```

3.2.1.3 Selection criteria

We have established two inclusion criteria (IC) and six exclusion criteria (EC):

- IC-1: The study elaborates on the use of ML for recommending refactoring.

- IC-2: The research is published in English.

The exclusion criteria used were are follows:

- EC-1: The study does not address Machine Learning;
- EC-2: The study does not address Recommendation;
- EC-3: The study does not address Refactoring;
- EC-4: The study is a secondary study;
- EC-5: The study is not available;
- EC-6: The study is an Abstract, poster, technical report, thesis, book, conference review, or patent.

3.2.1.4 Quality assessment

The quality of publications was measured after the final selection process. The following checklist was used to assess the credibility and thoroughness of the selected publications.

The quality of the selected publications was assessed after the final selection stage, using a structured checklist designed to evaluate the credibility, clarity, and methodological rigor of each study. The following criteria were used:

1. Does the paper have a well-defined approach?
2. Are the refactorings addressed or proposed by the approach explicitly described?
3. Is the Machine Learning classifier or model clearly defined and explained?
4. Does the paper provide empirical or theoretical validation of its findings?

Each publication received a quality score based on these four questions. For each criterion, a score of 0, 0.5, or 1 was assigned, reflecting whether the aspect was not addressed, partially addressed, or fully addressed, respectively. The total score

was obtained by summing the individual values, allowing a maximum score of 4 per publication. The complete results of the quality assessment are available in the replication package (GUISELLA et al., 2024).

3.2.1.5 Search Engines Databases

We select the most relevant online database chosen due to their popularity in software engineering publications. Table 1 shows the selected databases and the number of papers retrieved. It is noticed that Scopus Database returns a greater number of articles compared to the others.

Table 1 – Online Database

Source	URL	Retrieved
ACM	http://dl.acm.org/	17
IEEE	http://ieeexplore.ieee.org/	24
Science Direct	http://www.sciencedirect.com	13
Scopus	https://www.scopus.com/	121
WILEY	http://onlinelibrary.wiley.com/	2
		177

3.2.2 Data Extraction and Execution

The following information was extracted from the final set of papers:

- Papers metadata: title, authors, publication venue, year, pages, volume, abstract and document type;
- Refactoring researched in the approach;
- Machine Learning techniques employed;
- Dataset characteristics and feature details;
- Evaluation strategy of the approach;
- Automation level of the approach;
- Quality of the Recommendation.

3.2.3 Conducting the Review

The conducted process is shown in Figure 4, starting with running the search string (developed in Section 3.2.1.2) in the search engine databases (see Table 1). As a result, 177 papers were retrieved. Then, duplicate papers were removed. After that, non-relevant papers were filtered out by applying the inclusion/exclusion criteria, resulting in 26 papers. After reading the full papers, five papers were removed, leaving 21 papers. Finally, the snowballing technique was performed on the final selection, adding six more papers. Snowballing refers to the use of the reference list of a paper or the citations to the paper to identify additional sources (WOHLIN, 2014). For this Systematic review, forward and backward snowballing was used. Finally, 27 papers were selected for analysis and synthesis.

Table 2 shows the 27 papers, the result of the selection process. The table presents the paper ID and approach in the first and second columns. The third column shows the paper title, followed by the authors in the fourth column. In the fifth column, the Year of publication is listed, the type of refactoring research is in the sixth column; and the venue is in the last column. The analysis of the papers was conducted considering *Approaches* rather than the papers individually. This was done because some papers belong to the same research group, being just an evolution of the same approach. In this case, we grouped them under a unique Approach (referred to as A#) - see the first column of Table 2. Therefore, although there are 27 papers in the final set, we have 22 approaches. The papers [S04], [S12], and [S19] were grouped as the approach [A04], the papers [S05] [S17] and [S20] were grouped as the approach [A05] and papers [S18] and [S21], were grouped as [A16].

Note that, in the sixth column, the types of recommendations are classified as: i) unique refactoring recommendation (UniR) and ii) sequence of refactoring recommendations (SeqR). The first category consists of recommendations that suggest just one refactoring at a time. The second category offers a sequence of them, for example, *Apply the following refactorings: Extract Class, after Extract Method, and after Move Method.*

The approach [A01] introduces REMS, which recommends a unique refactoring, the Extract Method, to improve the internal structure of the method, solving code smells. The proposed process involves three steps: (1) extracting code property graphs from

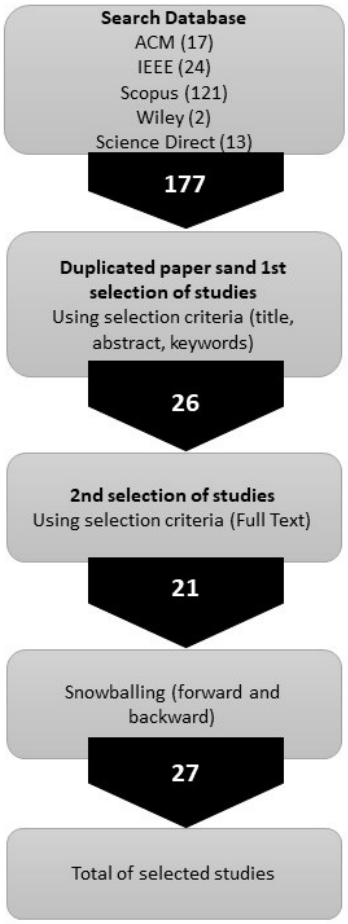


Figure 4 – Systematic Literature Review Process Phases

training and testing samples, (2) generating tree-view and flow-view representations using embedding techniques, and (3) training machine learning classifiers to recommend code fragments for extraction. The input of the approach includes Method source code, while the output is a set of recommended refactoring candidates. The approach has a principal limitation: its reliance on specific datasets, which may contain potential noise in the training data (as noted by the authors) and a lack of a user-friendly GUI.

The approach [A02] proposes a deep learning-based method that recommends a unique refactoring. It is focused on identifying extract method refactoring candidates. The process involves dataset preparation using RefactoringMiner and PyDriller, embedding generation with GraphCodeBERT to capture syntactic and semantic code

features, dimensionality reduction using an autoencoder, and training a Random Forest classifier for prediction. According to the authors, the main limitations lie in the dataset construction process and the quality of the negative samples.

The approach [A03] introduces AntiCopyPaster, an IntelliJ IDEA plugin that recommends a unique refactoring, the Extract Method, to avoid the introduction of duplicate code into the source code. The process involves detecting duplicate code fragments, extracting 78 metrics, and employing a CNN-based binary classifier to determine whether refactoring is needed. Inputs include pasted code fragments and their corresponding metrics, while outputs are refactoring recommendations presented via IDE notifications. As a limitation, the authors mention the potential overlap between positive and negative examples in the dataset, which can degrade the quality of the ML model.

The approach [A04] aims to prepare the source code of the software for the inclusion of new requirements. The process involves four steps: (1) data acquisition (collecting commit messages and feature requests), (2) text preprocessing (tokenization, stopword removal, and lemmatization), (3) feature modeling (numerical representation using TF-IDF), and (4) classifier training (to identify refactoring-related commits and predict specific refactoring types). Inputs include commit messages, feature requests, and historical refactoring data, while outputs consist of predictions on whether refactoring is needed and recommendations for specific sequence refactoring. A key limitation is the inability to precisely identify the specific code segments associated with the recommended refactoring.

The approach [A05] proposes an interactive approach, DOIMR, to improve refactoring recommendations by enabling developers to explore both objective spaces (quality metrics) and decision spaces (code locations) to enhance the quality attributes of QMOOD (reusability, flexibility, understandability, extendibility, and effectiveness) of the software. The process involves four steps: (1) multi-objective search to generate non-dominated solutions, (2) clustering solutions in the objective space based on quality attributes, (3) clustering solutions in the decision space based on code locations, and (4) integrating developer feedback to refine recommendations iteratively. The inputs are the source code and developer preferences, while the outputs are refactoring recommendations based-on the user preferences to improve specific quality attributes. The main limitation is that the solution requires an experienced user who understands clustering

and can navigate through the available options, which can be a tiring process.

The approach [A06] presents RMove, which recommends a unique refactoring, the Move Method refactoring, to remove the Feature Envy code smell. The construction process involves extracting structural information using Method Dependency Graphs (MDG) and semantic information using Abstract Syntax Trees (AST), generating hybrid embeddings, and training machine learning classifiers to suggest Move Method refactorings. The required inputs are the method and the class in which the method is located. According to the authors, the main limitation lies in the reliance on open-source datasets, which may introduce noise into the data.

The approach [A07] identifies the refactoring opportunity, recommending Class-level refactorings aimed at improving software quality. The process involves extracting 125 software metrics, applying multi-phased feature selection techniques (Wilcoxon rank-sum test, Pearson correlation test, PCA), addressing data imbalance through sampling methods (random, upsampling, downsampling), and performing classification using a heterogeneous ensemble structure with various machine learning algorithms. The output is a classification indicating whether the class needs to be refactored. The main limitation is the absence of an explicit indication of the refactoring type to be applied, with the approach identifying only the affected code elements, specifically, the class.

The approach [A08] recommends a unique refactoring for correcting flaws (functional decomposition) in UML class models. The goal of the approach is to improve design quality. The process involves transforming UML class diagrams into numerical representations, extracting design features, training a deep neural network to identify flawed designs, and applying refactoring operations to improve quality. The approach inputs include UML class diagrams and their metrics. The main limitation is the lack of standardized repositories for UML models and insufficient testing environments for model-level quality assurance.

The approach [A09] recommends a unique refactoring: removing relations, classes, and adding classes. These refactorings are applied to class diagram to address six different code smells at a higher level of granularity. The process involves extracting features from class diagrams and training a Bayesian network to recommend the appropriate refactoring. The required input is a class diagram in XMI format. A key limitation is the

need for labeled training data at the class level.

The approach [A10] recommends a unique refactoring to improve the quality of the source code. The approach focuses on the recommendation of the Extract Method in the context of ING, a large financial organization. The process involves analyzing Git history to identify refactored and non-refactored methods, extracting code metrics, training supervised machine learning models, and validating their predictions against expert opinions. The main limitation is that the study focuses on a single organization (ING), and the findings may not generalize to other industrial contexts or domains.

The approach [A11] shows an approach that recommends unique refactoring at the class, method, and variable levels. The goal is to investigate the effectiveness of machine learning algorithms in predicting software refactorings. The proposed approach involves three main steps: (i) data collection and feature extraction from over 11,000 open-source projects, (ii) training and testing six ML algorithms (e.g., Random Forest, Neural Networks) using refactoring and non-refactoring instances, and (iii) evaluating model accuracy, precision, and recall. Limitations include reliance on open-source projects, challenges in identifying non-refactored instances, and the need for fine-grained metrics and larger datasets for improved generalization.

The approach [A12] recommends a unique refactoring, the Move Method. The approach relies on the path-based representation of code for solving the Feature Envy code smell. The process involves identifying potentially movable methods, generating numerical embeddings for methods and target classes using the code2vec model, and employing a Support Vector Machine (SVM) classifier to recommend refactoring based on semantic similarity. Limitations include the reliance on the assumption that popular GitHub projects contain high-quality code and the reliance on the MoveMethodGenerator tool for creating a synthetic Dataset.

The approach [A13] proposes an approach to recommend a unique refactoring. The approach aims to solve duplicated code and long method code smells to improve the source code quality. The process involves extracting software metrics from five open-source projects in the Tera-PROMISE repository, applying data imbalance techniques (SMOTE, UPSAMPLE, RUSBoost), selecting significant features using the Wilcoxon rank test, and evaluating classifier performance in terms of AUC and accuracy. Limita-

tions include the partial information of the recommended refactorings and reliance on manually validated datasets.

The approach [A14] recommends classes to be refactored. This focuses on exploring the effectiveness of deep learning algorithms in building refactoring prediction models at the class level to improve software quality. The methodology involves preprocessing datasets, applying the GRU algorithm, and evaluating its performance. A key limitation is that the approach only identifies classes that require refactoring without specifying the type of refactoring to apply, resulting in an incomplete recommendation.

The approach [A15] presents an approach to recommend unique refactoring for solving code clone smells. The approach focuses on three types of refactorings: Move method, Pull up Method, and Extract Method. The process involves normalizing source code fragments, converting them into abstract syntax trees (ASTs), extracting features, detecting outliers using the Local Outlier Factor algorithm, and training classification models to predict refactoring recommendations. The main limitation is that, although the approach recommends three types of refactorings, it does not provide adequate guidance or identify the specific components that should be refactored.

The approach [A16], which is an approach that recommends methods to be refactored, focuses on exploring the effectiveness of ten supervised machine learning models in recommending methods that require refactorings to improve software quality. The process involves feature analysis using Wilcoxon rank-sum tests and logistic regression, data normalization, addressing class imbalance with techniques like SMOTE, UPSAMPLE, and RUSBoost, and training models using ten classifiers. The output is a binary classification indicating whether a method requires refactoring. The main limitation of the approach is that it does not specify the type of refactoring to be applied; it only identifies the methods that require refactoring.

The approach [A17] introduces a ML-based approach called CREC, that recommends a unique refactoring for solving code clones. The approach focuses on the recommendation of the Extract Method, and it points out the clones that need to be refactored, improving software maintenance. CREC operates in three phases: clone data preparation, feature extraction, and training/testing. The tool receives software repositories as input and outputs ranked recommendations for clone refactoring. The primary

concern identified by the authors relates to the quality of the dataset used for training.

The approach [A18] demonstrated a deep learning-based approach that recommends unique refactoring. This approach focuses on recommending the Move Method to identify and remove Feature Envy code smells. The approach generates a synthetic Dataset moving methods between classes in open-source applications, labeling them as smelly or non-smelly. As a result, the tool identifies smelly methods and recommends target classes for refactoring. The approach significantly outperforms state-of-the-art tools like JDeodorant and JMove in precision, recall, and accuracy. Limitations include reliance on high-quality training data, potential bias from generated smells, and a focus on misplaced methods rather than fields.

The approach [A19] recommends a unique refactoring, the Extract Method. This approach is based on probabilistic techniques for improving software maintainability. The input includes method-level software metrics from a single sample. The output is a ranked list of candidate refactorings. The main limitation is that, compared to other tools like JDeodorant, the approach generates a large number of candidates, which may overwhelm developers despite ranking efforts.

The approach [A20] introduces a tool called GEMS that recommends a unique refactoring. This approach provides a ranking of refactoring opportunities to apply the Extract Method, aiming to improve the quality attributes of the source code. GEMS learns a probabilistic model from open-source repositories, using structural and functional features that encode complexity, cohesion, and coupling. Input to GEMS includes Java methods, and its output is a ranked list of recommended code fragments for extraction. One limitation is its reliance on augmented training data, which may not fully represent real-world scenarios.

The approach [A21] recommends a unique refactoring of the Extract Method to improve software maintainability. The approach analyzes development histories to learn features of past refactorings and constructs a predictive model. The process involves detecting refactored and non-refactored methods, obtaining syntactic information, and constructing a learning model using tools like Weka. The proposed technique is not supported by a training tool that can be widely used by developers.

The approach [A22] recommends a unique refactoring to improve the structure of

the source code. The process involves using Least Squares Support Vector Machines (LS-SVM) with three kernel types (linear, polynomial, RBF), Principal Component Analysis (PCA) for feature extraction, and Synthetic Minority Over-sampling Technique (SMOTE) to handle imbalanced datasets. The main limitation is that it classifies classes requiring refactoring based on source code metrics and machine learning techniques. It doesn't directly target specific types of refactoring, but rather provides general examples of common refactorings that can be applied; therefore, the recommendation is incomplete.

The main findings and answers are presented in the following section.

Table 2 – The Final Set of Primary Studies

Approach ID	Paper ID	Paper Title	Authors	Year	Type	Venue
[A01]	[S01]	REMS: Recommending Extract Method Refactoring Opportunities via Multi-view Representation of Code Property Graph	Cui Di et al. (CUI et al., 2023)	2023	UniR	IEEE ICPC
[A02]	[S02]†	Automatic Refactoring Candidate Identification Leveraging Effective Code Representation	Palit et al. (PALIT et al., 2023)	2023	UniR	ACM
[A03]	[S03]	Just-in-time code duplicates extraction	AlOmar et al. (ALOMAR et al., 2023)	2023	UniR	Inf Softw Technol
[A04]	[S04]	Mining commit messages to enhance software refactorings recommendation: A machine learning approach	Nyamawe (NYAMAWE, 2022)	2022	SeqR	MLWA
	[S12]	Feature requests-based recommendation of software refactorings	Nyamawe et al. (NYAMAWE et al., 2020)	2020	SeqR	Empir. Softw. Eng.
	[S19]	Automated recommendation of software refactorings based on feature requests	Nyamawe et al. (NYAMAWE et al., 2019)	2019	SeqR	RE
[A05]	[S05]	Enabling Decision and Objective Space Exploration for Interactive Multi-Objective Refactoring	Rebai et al. (REBAI et al., 2020)	2022	SeqR	IEEE TSE
	[S17]	Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction	Alizadeh et al. (ALIZADEH et al., 2019)	2019	SeqR	SCAM

continuation of the previous page

Table 2 – continuation of the previous page

Approach	Paper ID	Paper Title	Authors	Year	Type	Venue
	[S20]	Reducing Interactive Refactoring Effort via Clustering-Based Multi-objective Search	Alizadeh et al. (ALIZADEH; KESSENTINI, 2018)	2018	SeqR	ASE
[A06]	[S06]	RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code	Cui Di et al. (CUI et al., 2022)	2022	UniR	ICSE
[A07]	[S07] †	Class-Level Refactoring Prediction by Ensemble Learning with Various Feature Selection Techniques	Panigrahi et al. (PANIGRAHI et al., 2022)	2022	Undef	Applied Sciences
[A08]	[S08]	A Machine Learning Approach to Software Model Refactoring	Brahmaleen et. al. (SIDHU et al., 2022)	2022	UniR	IJCA
[A09]	[S09]	A probabilistic-based approach for automatic identification and refactoring of software code smells	Saheb et al. (SAHEB-NASSAGH et al., 2022)	2022	UniR	Appl. Soft Comput.
[A10]	[S10]	Data-Driven Extract Method Recommendations: A Study at ING	Van der Leij et al. (LEIJ et al., 2021)	2021	UniR	ACM
[A11]	[S11]	The effectiveness of supervised machine learning algorithms in predicting software refactoring	Aniche et al. (ANICHE et al., 2020)	2020	UniR	IEEE
[A12]	[S13]	Recommendation of Move Method Refactoring Using Path Based Representation of Code	Kurbatova et al. (KURBATOVA et al., 2020)	2020	UniR	ICSEW
[A13]	[S14] †	Application of Naive Bayes classifiers for refactoring Prediction at the method level	Panigrahi et al. (PANIGRAHI et al., 2020)	2020	UniR	ICCSEA
[A14]	[S15] †	Harnessing deep learning algorithms to predict software refactoring	Alenezi et al. (ALENEZI et al., 2020)	2020	Undef	Telkomnika
[A15]	[S16] †	An Automatic Advisor for Refactoring Software Clones Based on Machine Learning	Sheneame (SHENEAMER, 2020)	2020	UniR	IEEE Access
[A16]	[S18]	Method Level Refactoring Prediction on Five Open Source Java Projects Using Machine Learning Techniques	Kumar et al. (KUMAR et al., 2015)	2019	Undef	ISEC

continuation of the previous page

Table 2 – continuation of the previous page

Approach	Paper ID	Paper Title	Authors	Year	Type	Venue
	[S21]	Application of SMOTE and LSSVM with various kernels for predicting refactoring at method level	Kumar et al. (KUMAR et al., 2018)	2018	Undef	ICONIP
[A17]	[S22]	Automatic Clone Recommendation for Refactoring Based on the Present and the Past	Yue et al. (YUE et al., 2018)	2018	UniR	ICSME
[A18]	[S23] †	Deep Learning Based Feature Envy Detection	Liu et al. (LIU et al., 2018)	2018	UniR	ASE
[A19]	[S24]	A log-linear probabilistic model for prioritizing extract method refactorings	Xu et al. (XU et al., 2017a)	2017	UniR	ICCC
[A20]	[S25]	GEMS: An Extract Method Refactoring Recommender	Xu et al. (XU et al., 2017b)	2017	UniR	ISSRE
[A21]	[S26]	Finding Extract Method Refactoring Opportunities by Analyzing Development History	Imazato et al. (IMAZATO et al., 2017)	2017	UniR	COMPSAC
[A22]	[S27]	Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level	Kumar and Sureka (KUMAR; SUREKA, 2017)	2017	Undef	APSEC

3.3 Answers to the Research Questions

3.3.1 Which relationships between Machine Learning and Refactoring Recommendations are explicitly discussed in the literature?

We divided RQ1 into three sub-questions: RQ1.1, RQ1.2 and RQ1.3.

3.3.1.1 RQ1.1 Which refactorings have been investigated?

To answer this research question, we considered the catalog proposed by Martin Fowler (FOWLER; BECK, 2019) which is composed of 67 code refactorings. This catalog served as a reference to identify the most and least investigated refactorings.

However, we observed that not all refactorings addressed in this systematic review are included in Fowler's catalog. Specifically, there are 22 refactorings proposed by Fowler and 10 by other authors (Non-Fowler).

Table 3 shows in the first column the refactorings; in the second, the number of papers; and in the last column, the references. Some refactorings have alternative names, so we group them and separate the names with a slash.

Answering the research question, Table 3 shows that *Extract Function* (aka *Extract Method*) and the *Move Function* (aka *Move Method*) are the two most investigated refactorings, explored by 13 and 8 approaches, respectively. According to the authors, the 'Extract Method' refactoring is chosen due to its widespread adoption in software development [A01, A02, A03, A10, A19, A21]. It is consistently highlighted as one of the most frequently performed and popular refactoring operations [A01, A02, A04, A21]. Additionality, since it decomposes large and complex methods, it addresses critical design flaws such as Duplicated Code, Long Method, and Feature Envy [A01, A03, A10, A19, A17]. This refactoring significantly improves code maintainability, comprehension, and reusability by clarifying and simplifying internal structures, thereby enhancing overall software quality [A01, A02, A03, A19, A21]. It can also be combined with other refactoring operations to achieve broader design improvements [A01, A03, A05].

In the case of the Move method, the authors noted its fundamental role in addressing the 'Feature Envy' code smell [A06, A12, A18]. This refactoring improves the internal structure of software, enhancing code maintainability and comprehension [A06, A12, A18]. In addition, the authors also selected 'Move Method' because it is one of the most frequently performed and widely adopted refactoring operations [A05, A06, A12, A18].

In addition to the most frequently investigated refactorings, it is also important to consider the least explored ones. Out of Fowler's catalog of 67 refactorings, 45 did not appear in our final set, which accounts for 67% of the catalog. If we expand this analysis by counting the refactorings that do not appear as well as those that appear just once (nine in total), this percentage rises to 81%, corresponding to 54 refactorings from 67. Therefore, a significant amount of effort is concentrated on just two or three refactorings, while the rest are left out.

Table 3 – Refactorings addressed by the selected papers

Fowler Refactorings		
Refactorings	**#N**	**Approaches (A#)**
Extract Function/ Extract Method	13	[A01],[A02],[A03], [A04], [A05], [A10], [A11], [A13], [A15], [A19], [A17], [A20], [A21]
Move Function/ Move Method	8	[A04],[A05], [A06], [A08],[A11], [A12], [A15], [A18]
Pull Up Method	6	[A04], [A05], [A08],[A11], [A13], [A15]
Extract Superclass	4	[A04], [A05], [A08], [A11]
Push Down Method	4	[A04], [A05], [A08], [A11]
Rename Function /Rename Method	4	[A04], [A05], [A08], [A11]
Move Field	3	[A04], [A05], [A08]
Pull Up Field	3	[A04], [A05], [A08]
Push Down Field	3	[A04], [A05], [A08]
Extract Class	3	[A05], [A08], [A11]
Replace Type Code with Subclasses / Extract Subclass	3	[A05], [A08], [A11]
Encapsulate Variable /Encapsulate Field / Self-Encapsulate Field	2	[A04], [A05]
Inline Function	2	[A04], [A11]
Rename Field	1	[A08]
Introduce Parameter Object	1	[A13]
Preserve Whole Object	1	[A13]
Replace Function with Command / Replace Method with Method Object	1	[A13]
Replace Temp with Query	1	[A13]
Substitute Algorithm	1	[A13]
Extract Variable	1	[A11]
Inline Variable	1	[A11]
Rename Variable	1	[A11]
Refactorings proposed by other authors (non-Fowler Refactorings)		
Refactorings	**#N**	**Approaches (A#)**
Rename Class	2	[A04], [A11]
Extract Interface	2	[A04], [A11]
Move Class	2	[A08], [A11]
Increase-Decrease Field Security	1	[A05]
Increase-Decrease Method Security	1	[A05]
Extract Associated Class	1	[A08]
Decomposition objects	1	[A07]
Removing Relations	1	[A09]
Removing Classes	1	[A09]
Adding Classes	1	[A09]

It is difficult to determine precisely why some refactorings are little researched. Possible reasons include the lack of popularity of such refactorings and the lack of me-

trics/tools able to detect them. Regardless of the reason, it is clear that some refactorings are more challenging to recommend than others. For example, the refactorings *Rename Class* and *Introduce Parameter Object* have been very little investigated. In the case of *Introduce Parameter Object*, the reason for its limited investigation may be the lack of low-level metrics (parameter level) able to detect/characterize the problem. Such analysis should reveal that a set of method parameters should be encapsulated in a new class. This requires a semantic analysis to discover which parameters are significant and whether it makes sense to include them as fields of a new class; an inherently challenging task.

The approaches [A07], [A14], [A16], and [A22] were omitted from Table 3 because they do not provide specific Fowler or non-Fowler refactorings. These approaches identified components (Classes or Methods) and recommended them for refactoring. This implies that such recommendations are incomplete, as they fail to specify which refactoring should be applied.

3.3.1.2 RQ1.2 Which ML algorithms have been used to recommend refactorings?

Table 4 presents the ML algorithms that appear in the final set of papers. Remarkably, the most employed learning process is Supervised Learning with 28 algorithms. One reason for its predominance is its higher predictive accuracy and interpretability, especially in classification tasks, which are traditional in refactoring prediction.

Answering the RQ1.2, the most used algorithms are: Random Forest (RF) researched by 11 approaches; Logistic Regression (LR) researched by 9 approaches; and Support Vector Machine (SVM) researched by 8 approaches. Note that some approaches have employed more than one algorithm. For example, the approach [A01] makes use of 9 different supervised learning algorithms such as Random Forest (RF), Logistic Regression (LR), Support Vector Machine (SVM), Naive Bayes, among others.

Random Forest (RF) algorithm is frequently chosen by authors due to its consistently high performance and accuracy across diverse refactoring prediction tasks [A02, A10, A11, A15, A16, A17, A21], as well as in software engineering problems such as code smell identification [A02, A10, A11]. RF is recognized for its robustness and stability, often outperforming other machine learning algorithms and maintaining strong predictive capabilities even when generalized to different contexts or projects [A06, A10,

Table 4 – Algorithms used by the selected papers

L.P.	Algorithm	Qty	Approach
Supervised	Random Forest (RF)	11	[A01], [A02], [A04], [A06], [A10], [A11], [A15], [A16], [A17], [A19], [A21]
	Logistic Regression (LR)	9	[A01], [A04], [A16], [A06], [A07],[A10], [A11], [A20], [A22]
	Support Vector Machine (SVM)	8	[A01], [A04], [A06], [A10], [A11], [A12], [A20], [A22]
	Naïve Bayes (NB)	7	[A01], [A16], [A06], [A10], [A11], [A13], [A17]
	Decision Tree (DT)	6	[A01], [A04], [A06], [A07], [A10], [A11],
	Least-squares support-vector machine (LSSVM)	4	[A16], [A07], [A22], [A22]
	K-nearest neighbors (KNN)	4	[A01], [A07], [A15], [A20]
	Bayesian Network (BN)	4	[A09], [A16], [A22], [A22]
	Gradient Descent (GD)	3	[A16], [A06], [A08]
	Gradient boosting classifier	2	[A19], [A20]
	AdaBoost	2	[A16], [A17]
	Extreme Gradient Boosting	2	[A01], [A06]
	Multinomial naive bayes (MNB)	1	[A04]
	Bagging	1	[A15]
	LogitBoost	1	[A16]
	J48	1	[A22]
	C4.5	1	[A17]
	Sequential minimal optimization	1	[A17]
	ForestPA	1	[A15]
	Convolutional Neural Network [1]	6	[A01], [A03], [A04], [A06], [A11], [A18]
	Artificial Neural Network (ANN)[1]	2	[A16], [A07]
	Levenberg Marquardt Algorithm[1]	2	[A16], [A06]
	Radial Basis Function Network [1]	2	[A16], [A06]
	Gated Recurrent Units Recurrent NN (GRU) [2]	2	[A06], [A14]
	Long Short Term Memory Recurrent Neural Network (LSTM) [2]	2	[A01], [A06]
	Deep neural network model [2]	1	[A08]
	Extreme Learning Machine [2]	1	[A07]
	Multilayer Perceptron [1]	1	[A16]
Un-superv	Clustering Ensembles with Pareto-front	1	[A05]
	Density-based algorithm	1	[A15]
	Clustering + Algorithm NSGA-II	1	[A05]
	Clustering + Genetic Algorithm	1	[A05]

[1] Artificial Neural Network
[2] Deep Learning Models

A11, A17]. Its ensemble learning nature (combining multiple decision trees with random data subsets) allows it to learn complex, non-linear relationships within the data [A02, A11, A15]. Furthermore, RF handles high-dimensional and sparse datasets effectively [A04].

Logistic Regression (LR) algorithm is also commonly chosen by authors due to its widespread use [A01, A04, A06, A07, A11, A16]. Despite its simplicity, LR can outperform Naive Bayes in refactoring prediction [A11, A16]. LR models generalize well across datasets [A11] and their log-linear probabilistic nature allows the integration of diverse software metrics [A19]. Additionally, LR is well-suited for handling high-dimensional and sparse data, which are common characteristics in software engineering problems, and it generally demonstrates good accuracy in predicting refactoring opportunities [A04, A11].

3.3.1.3 RQ1.3 How datasets and features can be classified?

To answer this research question, we conducted two analyzes. The first provided a classification for the datasets, and the second offered a classification of the features used to compose the datasets. Together, these analyzes help to understand the diversity and characteristics of data sources and feature sets employed in refactoring recommendation research, highlighting both common practices and gaps in the literature.

Table 5 presents the results of our analysis. The first column lists the approach; columns two through six show the feature groups; the seventh column indicates the metric extraction tool; the eighth column specifies the type of labeling; and the last column describes the dataset type. Regarding the dataset type, we identified two types:

- *Code Smell-based (CS)*: The instances or samples of the dataset are methods or classes identified as having a code smell. These instances are extracted from the current version of the projects, without considering previous commits. Models trained on this type of dataset can identify only code smell issues.
- *Refactoring-based (RB)*: The instances are methods or classes that have undergone refactorings in past commits. These instances are extracted from the project history using mining tools. Models trained on RB datasets are expected to identify a broa-

der range of refactoring opportunities, including code smells and other situations where refactoring might be applicable.

From Table 5, it is evident that 15 approaches (representing more than 68%) use RB-datasets. This choice is based on the underlying assumption that refactorings performed in the past serve as good examples of refactorings that will be necessary in the future. Several approaches [A01], [A10], [A11], [A17], [A19], [A21] note that historical refactoring enables machine learning models to learn complex relationships and interactions between code features and refactoring decisions that are difficult to capture through static rules.

A notable finding is that 9 of the 15 approaches explores the history of the projects to recommend Extract Method refactoring [A01], [A02], [A04], [A10], [A11], [A17], [A19], [A20] and [A21]. The approaches highlight that analyzing past refactorings allows them to gain empirical insights into why specific refactorings were performed and enables the development of recommendations that better reflect real developer practices.

The second analysis focused on classifying the features of the datasets. Table 5 shows the feature categories (`C1` to `C5`) used in the datasets. We identified five distinct types:

- **`C1` – Model Metrics**. Metrics extracted from UML diagrams and graph models, such as Probabilistic Graphical Models (PGMs). Examples include: for UML, the number of generalizations, associations, and classes; for PGMs, the depth of the inheritance tree and the number of relations between classes.
- **`C2` – Source Code Metrics**. It is about the well-known metrics of complexity, coupling and cohesion. For example: Lines of code (LOC), response for class (RFC), and coupling between objects (CBO);
- **`C3` – Tool-based information**. Data extracted from tools such as issue trackers (e.g., Jira summaries, descriptions, and status) and version control platforms (e.g., number of commits, commit messages, and dates from GitHub);

Table 5 – Dataset features by the selected papers

App.	C1	C2	C3	C4	C5	Metric Extraction Tool	Lbl.	D.T.
[A01]	✓					NA	NA	RB
[A02]				✓		Autoencoder technique	A	RB
[A03]		✓				Non-specified	A	RB
[A04]			✓			Python Natural Language Processing Toolkit/Codacy tool	A	RB
[A06]		✓		✓		SRCML, ASTMiner and DEPENDS tools	A	CS
[A07]		✓				NA	NA	RB
[A08]	✓					SDMetrics tool	M	CS
[A09]	✓					NA	NA	CS
[A10]		✓				NA	NA	RB
[A11]		✓	✓			SourceMeter	A	RB
[A12]		✓		✓		code2vec technique	A	CS
[A13]		✓				NA	NA	RB
[A14]		✓				NA	NA	RB
[A15]		✓				Java Development Tool (JDT)	NS	CS
[A16]		✓				NA	NA	RB
[A17]		✓			✓	MCIDiff	M	RB
[A18]		✓		✓		Distance Metric and Word2vector Technique	A	CS
[A19]		✓				Non-specified	M	RB
[A20]		✓				Proprietary Algorithm for Extracting code features	A	RB
[A21]		✓				H. Murakami technique	A	RB
[A22]		✓				NA	NA	RB

Lbl → A: Automated; M: Manual; M: Manual; NS: Non-Specified; NA: Non-Applied
D.T. → CS: Code Smell - based; RB: Refactoring - based

- `C4` **– Semantic information Metrics**. It comprises metrics related to the extraction of relationships between the source code. For example, the relation between the method names and class names;

- `C5` **– Project History Metrics**. Metrics that reflect the historical evolution of a project. The intuition is that a project's evolution history may imply its future evolution. For example: 'a percentage of change commits among all commits'.

It is remarkable that `C2` (Source Code Metrics) emerges as the most prevalent

category with 16 out of 22 approaches, representing more than 72%. We believe this occurs because object-oriented metrics are easy to extract and automate with static analysis tools. Additionally, there is a long history of using these metrics, many public datasets available, and clear definitions that help researchers reproduce and compare results across studies. Regarding the `C5` group, only the approach [A17] uses the information about the history of the project. The approach defines 6 metrics to capture the evolution history of individual clones, reflecting how their past changes may influence developers' refactoring decisions.

Table 5 also includes the columns *Metric Extraction Tool* and *Label*, which highlight the tools used and the additional effort invested in building each dataset. The value "NA" (not applicable) indicates that authors used ready-to-use datasets from the literature rather than extracting features themselves. For example, [A01] uses the Silva dataset (SILVA et al., 2016) and the Xu dataset (XU et al., 2017b); [A09] uses the Fontana dataset (FONTANA et al., 2016) and the Palomba dataset (PALOMBA et al., 2018); [A10] reuses part of the dataset from Aniche et al. (ANICHE et al., 2020). The approaches [A07], [A13], [A14], [A16], and [A22] rely on the Tera-Promise dataset, a widely used repository that contains source code metrics and refactorings extracted from two successive releases of seven open-source Java projects.

3.3.2 RQ2 - Does the way the approaches are evaluated depend on the automation level of them?

This question is broken down into two sub-questions, RQ2.1 and RQ2.2, aiming to describe the two main concepts involved: the evaluation method and the automation level of the approaches. So, Table 6 was elaborated to summarize how the approaches conducted their evaluations. The first column lists the Approach ID; columns two to five indicate the evaluation types (I, II, III, or IV); the sixth column specifies the abstraction level of the evaluation; the seventh presents the results only for evaluations II and III; the eighth outlines the limitations identified; and the last column reports the automation level of the approach.

3.3.2.1 RQ2.1 - How have the approaches been evaluated?

Various evaluation strategies are employed to evaluate the approaches. The following categories summarize the four types of evaluation methods used by the papers in this systematic review to benchmark and validate their approaches.

- **(I) Comparison with other state-of-the-art approaches/tools**. This evaluation consists of comparing the performance of the approach with other similar approaches and/or tools;
- **(II) Evaluation of the classifiers**. This is a type of internal evaluation where the goal is to identify the best classifier among those used in the approach;
- **(III) Evaluation of one Classifier**. Another kind of internal validation. In this case, the authors evaluated the performance of the only classifier used in the approach;
- **(IV) Controlled Experiment**. This evaluation involves a group of participants whose goal is to evaluate the usefulness of the approach/tool when compared with other approaches/ tools.

The results of the evaluation analysis are shown in Table 6. Regarding evaluation type I, which consists of comparing the proposed approach with other state-of-the-art tools or methods, we observed that 12 approaches ([A01], [A02], [A04], [A05], [A06], [A09], [A12], [A15], [A17], [A18], [A19], and [A20]) adopt this evaluation strategy. Most of these evaluations are empirical and quantitative, relying on objective performance metrics calculated automatically from datasets, without involving user participation or qualitative feedback.

To better understand the tools employed in these comparative analyzes, Table 7 reports the distribution of several well-known tools as well as lesser-known ones (for which we included the authors' names for identification). Notably, JDeodorant (FOKAEFS et al., 2007) stands out as the clear standard in this context, being used in 7 out of the 12 studies (almost 60%). This predominance is likely due to its maturity, availability, and broad functionality, capable of detecting five types of code smells and addressing them through recommended and automated refactorings. Other relatively

popular tools include JMove (TERRA et al., 2018), JExtract (SILVA et al., 2015), and SEMI (CHARALAMPIDOU et al., 2016).

Beyond this, Table 7 reveals a long tail of tools that appear only once. This dispersion suggests that several studies rely on in-house or ad-hoc implementations created to address very specific research objectives. While such tools may effectively serve the purposes of their respective studies, their narrow scope and limited documentation make it challenging to transform them into reusable solutions for the community.

In relation to evaluation types II and III, which correspond to the internal validation of classifiers as described above, Table 6 shows that 14 approaches, representing 73% of the total, applied these types of evaluations. Analyzing the results reported by these papers, we found that the Random Forest algorithm consistently achieved the best performance, followed by SVM and Naive Bayes, despite variations in evaluation conditions and datasets. This consistency suggests that Random Forest is a robust choice for the classification tasks under study. However, it is worth noting that differences in datasets and experimental setups across the papers may affect the generalizability of these findings.

Regarding type IV, controlled experiments, only five approaches [A01], [A03], [A05], [A06] and [A10] conducted this evaluation method. These experiments focused on evaluating the usability and effectiveness of the proposed tool/approach. In the case of [A01], the authors conducted a study with 10 experienced participants to evaluate the usefulness of the tool called REMS. Participants were presented with refactoring recommendations generated by different tools and were asked to evaluate these recommendations and complete a questionnaire. In the case of [A03], the authors provided a plugin for the AntiCopyPaster tool along with a video demonstrating its usage. Participants were then surveyed using 21 questions to assess the tool's usefulness, usability, and functionality.

In the evaluation in [A05], users' perceptions of how meaningful the recommended refactorings were measured. Additionally, it recorded the time developers spent identifying the best refactoring strategies, as well as the number of interactions during the process. For [A06], participants were given the source code of the FreeMind project alongside a list of refactoring solutions generated by four different tools. They were

then asked to complete a questionnaire regarding the refactoring tools and the respective solutions. Finally, in [A10], the authors designed a 30-question survey and asked five senior engineers to decide whether to refactor the given methods.

Table 6 – Evaluation and automation of approach in the selected papers

ID	I	II	III	IV	Level	Evaluation Result (II and III)	Limitation	Auto
[A01]	✓	✓		✓	Method	The KNN outperformed the other 8 classifiers. The combination with CodeBERT embedding technique shows a better performance	The small size of the dataset can compromise the result of the trained model.	Semi-Aut
[A02]	✓				Method	The comparison was against the state-of-the-art approach from Aniche et al. [11], taking as baseline the random forest model.	Other existing approaches in the literature are not used for comparison.	Semi-Aut
[A03]		✓		✓	Method	Convolutional Neural Network, Random Forest and Support Vector Machine are the models with better performance	The authors only use Convolutional Neural Network (CNN) for their propose approach.	Fully-Aut
[A04]	✓	✓			Class/ Method/ Attr.	For binary Classifier MNB outperformed the rest of the classifiers. For Multi-label Classifiers, SVM had the best performing classifier.	The proposed approach is compared with its previous version	Semi-Aut
[A05]	✓			✓	Class/ Method/ Attr.	The authors conduct a experiment where developers manually evaluate solutions to estimate the relevance of refactorings.	No measures or internal quality indicators for estimating the relevance of refactorings were used.	Fully-Aut
[A06]	✓	✓		✓	Method	The authors train classifiers with various embedding techniques. The Code2Vec+SDNE (CV+SN) embedding technique with NB (Naive Bayes) was the most effective combination.	Deep learning classifiers do not perform as well as the authors expected in recommending move method refactorings. The authors believe this is cause by type of the data.	Semi-Aut
[A07]		✓			Class	The MVE (maximum voting ensemble) with upsampling shows a better performance when compared with the other classifiers in the proposed approach.	PROMISE is a well-known and widely used dataset in this context. A comparative evaluation with other approaches/tools could be conducted to demonstrate the performance.	Semi-Aut

continuation of the previous page

Table 6 – continuation of the previous page

ID	I	II	III	IV	Level	Evaluation Result (II and III)	Limitation	Auto
[A08]			✓		Class (Model)	The authors show that with the use of deep neural network it is possible to detect models as flawed by functional decomposition (FD) with a precision of 0.87.	The evaluation focused on the identification of the FD and does not make it clear how to apply the recommended solution, what is the order of application and where it should be applied.	Semi-Aut
[A09]	✓				Class	The approach was compared with the work of Di Nucci et al.	The authors only use Bayesian networks model.	Semi-Aut
[A10]				✓	Method	The survey consists of 30 questions. Additionally, the authors performed a comparison between Datasets.	The paper is a case study within a single organization, ING, a large financial organization.	Semi-Aut
[A11]		✓			class, method, and variable-levels	Random Forest has the highest overall accuracy among all the 6 models	The approach is only intended to compare the models, and not to solve any problem or provide any recommendations.	Semi-Aut
[A12]	✓				Method	Datasets: JMove's dataset and MoveMethodDataset	The authors only use SVM classifier.	Semi-Aut
[A13]			✓		Method	Three Naïve Bayes classifiers were used (GNB, MNB, BNB) and the BNB presented the best performance in terms of AUC and Accuracy.	In this study, the author only use Naive Bayes and do not experiment with other classifiers.	Semi-Aut
[A14]			✓		Class	Compares the performance of GRU classifiers using balanced and imbalanced datasets. The balanced one had the best performance.	In this study, the author only use GRU and do not experiment with other classifiers.	Semi-Aut
[A15]	✓	✓			Method	ForestPA and RF achieved the best results among all the classifiers.	The Dataset used for evaluation are small.	Semi-Aut
[A16]		✓			Method	AdaBoost and ANN+GD classifiers outperformed the other classifiers. In addition, the authors show that using balance techniques can produce statistically significant differences in performances.	PROMISE is a well-known and widely used dataset in this context. A comparative evaluation with other approaches/tools could be conducted.	Semi-Aut
[A17]	✓	✓			Method	AdaBoost suggests clones for refactoring with high accuracy.	Comparison with an only one approach developed in 2014.	Semi-Aut
[A18]	✓				Method	The destination part of the recommendation was evaluated, i.e., the correct identification of the target class.	The Dataset was generated artificially, i.e., all the Feature Envy smell were made moving the methods manually.	Semi-Aut

continuation of the previous page

Table 6 – continuation of the previous page

ID	I	II	III	IV	Level	Evaluation Result (II and III)	Limitation	Auto
[A19]	✓				Method	Dataset: 5 open source software projects	Dataset small, composed by only 267 Extract Method instances.	Semi-Aut
[A20]	✓	✓			Method	GB classifier presents better performance of the others.	The Dataset used for evaluation was small - only 267 instances.	Semi-Aut
[A21]		✓			Method	SVM has high Precision, but lower Recall. On the other hand, the algorithms based on decision tree (J48 and RandomForest) record over 89% for both of Precision and Recall.	There are other approaches/tools in the literature that recommend the "Extract Method"and it would have been interesting to observe the performance of the proposal compared to them.	Semi-Aut
[A22]			✓		Class	The authors demonstrated that LS-LSM RBF kernel variant outperforms linear and polynomial kernel.	In this study, the author only one classifier Least Squares Support Vector Machines (LSSVM).	Semi-Aut

Table 7 – Tools used in evaluation type I

		#Papers	Approach
Tool Name	JDeodorant (FOKAEFS et al., 2007)	7	[A01], [A05], [A06], [A12], [A18], [A19], [A20]
	JMove (TERRA et al., 2018)	3	[A06], [A12], [A18]
	JExtract (SILVA et al., 2015)	3	[A01], [A19], [A20]
	SEMI (CHARALAMPIDOU et al., 2016)	2	[A01], [A20]
	PathMove	1	[A06]
	GEMS	1	[A01]
	Segmentation	1	[A01]
Unknown tool	Wang and Godfrey	2	[A15][A17]
	Charalampidou et al.	1	[A04]
	Ouni et al.	1	[A05]
	Mkaouer et al.	1	[A05]
	Alizadeh et al.	1	[A05]
	FR-Refactor (Nyamawe et al.)	1	[A04]
	CREC (Yue et al.)	1	[A15]
	Di Nucci et al.	1	[A09]
	Aniche et al.	1	[A02]

3.3.2.2 RQ2.2 - What is the automation level of the approaches?

To classify our final set of papers, we based our taxonomy on the works of (ALIZADEH et al., 2019) and (SIMMONDS; MENS, 2002), while extending their original classification. Our classification considers the following levels:

- **Fully automated**: The approach involves a tool capable of identifying refactoring opportunities, providing recommendations to the developer, and allowing the developer to apply the refactoring directly.
- **Partial/semi-automated**: The approach lacks a fully integrated tool within an IDE. Instead, the refactoring process is carried out in multiple steps or phases, often relying on trained classifiers.

The last column of Table 6 shows the automation levels of the approaches. Accordingly, 20 out of the 22 approaches provide no direct support for guiding the developer through the refactoring process, indicating that approximately 90% of the approaches are semi-automated. The only two fully automated approaches are [A03] and [A05], both of which assist in identifying refactoring opportunities. Specifically, [A03] introduces the AntiCopyPaster plugin, which monitors the introduction of potentially duplicate code and recommends refactorings via the IDE's Extract Method feature. Meanwhile, [A05] presents the DOIMR tool, which interacts with developers by displaying a list of refactoring recommendations that can be evaluated and selected according to the developer's preferences.

To better understand the reasons behind the low availability of automated refactoring tools, we present Table 8, which summarizes the main challenges and limitations reported by the approaches concerning the implementation of automatic refactoring tools. The first column, Category, groups the issues into four broad dimensions: Data and Modeling, Performance, Formulation of the recommendation, and Usability and Adoption. The second column, Description of Challenge or Limitation. The third column, References, lists the approaches that report each limitation.

The Table 8 highlights key challenges and limitations, including data quality issues, a lack of reliable negative samples, and data imbalance. High computational costs

Table 8 – Challenges and limitations for implementing tool support

Category	Description of Challenge / Limitation	References
Data and Modeling	**Data quality and generalization**. Training data with noise, unverified assumptions about quality, evaluation limited to few projects/datasets, difficulty generalizing to industrial projects or other communities.	[A02], [A04], [A10], [A12]
	Difficulty generating negative samples. Approximate heuristics that may introduce noise and lack of reliable datasets with exemplary code examples.	[A02], [A10], [A11]
	Imbalanced data. Need for techniques such as undersampling, lack of reliable estimates of real-world distribution.	[A11], [A13], [A16], [A22]
Performance	**High computational costs**. Training and processing require significant resources, hindering large-scale use and IDE integration.	[A02], [A10], [A12]
	Dependence on external tools. Approaches use for mining repositories, metrics extraction, creation of synthetic dataset, etc. Introducing noise.	All the approaches
Formulation of the Recommendation	**Low granularity recommendations**. Models identify methods/ Classes but do not specify all components involved for a complete recommendation.	[A02], [A04], [A07], [A08], [A10], [A11], [A13] ,[A14] , [A15] , [A16]
	Lack of consideration for human factors and context. Models relying only on code metrics ignore real developer motivations, subjectivity, and experience.	[A04], [A08], [A10], [A11], [A19], [A22]
	Recommendation prioritization. Excessive suggestions can overwhelm developers without effective ranking mechanisms.	[A10]
Usability and Adoption	**Low tool adoption**. Automatic refactoring tools have not been widely adopted due to frequent false positives and low developer trust.	[A11], [A19]
	Interpretability of "black-box"models. Difficulty understanding why a recommendation was formulated is a barrier to trust and accept it.	[A10], [A13], [A22]
	Deployment issues. High disk and memory requirements prevent direct IDE integration; centralized servers add complexity.	[A10]
	Need for human validation. Lack of validation by real developers; difficulty recruiting them for manual verification.	[A04]

and reliance on external tools hinder scalability and integration. Recommendations often lack granularity and overlook human factors like developer motivations and context. Usability is affected by frequent false positives, difficulty interpreting black-box models,

and deployment barriers due to resource demands. Low adoption rates reflect these technical and human challenges, indicating that current tools still struggle to meet practical developer needs effectively.

Overall, the results reveal a clear gap between the approaches and the availability of fully automated refactoring tools. The predominance of semi-automated approaches (around 90%). This scenario highlights that overcoming these obstacles is essential to enable higher levels of automation in refactoring recommendation tools and to bridge the gap between academic advances and practical adoption.

3.3.2.3 RQ3 - Have the approaches concerned with the quality of the recommendations?

This research question aimed to analyze the extent to which approaches have been concerned with the quality of the recommendations they provide from the developer point of view. We consider a high-quality recommendation to be one that provides developers with all the information they need to decide whether to accept the recommendation or not. In this work, we prefer to use the term *complete* for classifying the recommendations. We have devised our own definition of *complete* to enable a meaningful comparison between approaches. This was necessary because the approaches differ substantially in terms of recommendation quality. While some recommendations are quite complete, providing software engineers with detailed information, others lack detail, suggesting only the name of a refactoring or the location in the source code where the refactoring must be applied.

This analysis presents the development of a completeness criterion called *W3B*, which will be widely addressed in **Chapter 4**.

3.4 Discussion

Our results allow some topics worth further discussion to emerge:

Refactoring research - Extract Method - Table 3 shows that the three most researched refactorings are: Extract Method, Move Method, and Pull-up Method. The high interest in these refactorings may indicate their importance in the industrial sector and suggest

that these activities are more frequently applied in practice than others. In addition, we found that most of the refactorings proposed by Fowler (45 out of 67 refactorings) were not considered in any of the approaches in our final set. This result shows a gap between refactoring practice and research in the area of identifying refactoring opportunities.

Supervised learning techniques are favored over unsupervised learning techniques - This numerical superiority is due to the nature of recommendations. To recommend refactorings, it is necessary to learn how to identify *the opportunities for refactoring*. In the literature, extensive research has been conducted on this topic, exploring different indicators such as code smells and software quality attributes. Thus, there is a vast amount of metrics that can be used to set up a data repository.

The quality of the datasets - The quality of datasets is a major concern across the approaches because it strongly affects how reliable and generalizable refactoring recommendations are. All approaches agree that noisy or biased training data harms model performance, showing that data quality is often more important than quantity. A key challenge is creating good negative samples to keep the datasets balanced. Overall, the approaches share the view that careful data preparation and validation are essential to build robust and trustworthy automated refactoring tools.

The nature of the datasets impacts the refactoring opportunities - Regarding the type of Datasets (Refactoring-based (RB) and Code smell-based (CS)), we believe an important difference of the RB over the CS samples is the potentially valuable information that can be explored. Classifiers trained with refactoring-based datasets have the advantage of identifying a wider range of refactoring opportunities. However, this type of approach requires special treatment when building the dataset, since each instance must represent a snapshot before the application of a refactoring. This requirement makes data collection and processing more complex and resource-intensive.

Features explored in Datasets - The results reported in Table 5 shows that the most frequent group of features used to build the datasets is source code metrics. These features are clearly concentrated on static analysis, reflecting a limited exploration of other types of indicators. Researchers are encouraged to investigate additional sources of features, such as process metrics, historical data, or developer interaction data, and to evaluate whether models trained with such non-traditional features can improve predictive

performance.

Lack of mature tools - Overall, more than 90% of the approaches identified in our study are semi-automated, with only two papers proposing fully automated solutions with tool support. However, the adoption of these tools in the software engineering industry is hindered by a combination of technical, methodological, and practical challenges, as summarized in Table 8. These factors collectively explain the current low level of automation in refactoring tools and the gap between research prototypes and industry-ready solutions.

3.5 Threats to Validity

Internal validity: All relevant papers were retrieved using the defined search strings across major databases and through snowballing from reference lists. We covered five key publication venues (ACM, IEEE, Science Direct, Scopus and Wiley). However, there is a potential threat related to the formulation of the search strings, which might have excluded relevant studies that used different terminology. In addition, while the inclusion and exclusion criteria were clearly defined, their application inherently involves a degree of subjectivity that may influence the final set of selected studies.

External validity: The collected papers are primarily academic, which may limit the generalizability of our findings to industrial contexts. Although the academic literature provides a solid foundation for identifying trends, methods, and limitations, the actual applicability and impact of these approaches in real-world software development environments may differ.

Construct validity: The classification of approaches and extraction of data were based on our interpretation of the information provided in the papers. Incomplete or ambiguous descriptions in the original works may have affected the accuracy of our categorizations. Although cross-validation between authors was employed to mitigate misinterpretations, some degree of bias is still possible.

Conclusion validity: Our conclusions are based on the analyzed data but the dataset's scope and composition may limit its strength. Observed patterns indicate trends, not causation, and may change as new studies emerge.

3.6 Conclusion and Future Directions

This Systematic Literature Review reports the use of Machine Learning in the context of refactoring recommendations. A total of 177 potential articles were identified in five scientific digital libraries during the period from 2015 to 2023. After screening, our final set resulted in 27 papers, which we grouped into 22 approaches. Our findings reveal that research in this field is still concentrated on a limited subset of refactoring types, with several Fowler's refactorings largely overlooked. Moreover, our analysis highlighted that refactoring-based datasets tend to capture a wider range of opportunities compared to code smell-based datasets, potentially enabling more comprehensive models despite being more challenging.

For future research, we highlight several directions:

(1) The building of datasets is accompanied by detailed processes for their construction, including strategies for selecting negative samples. It has been observed that a robust dataset is fundamental for the proper training of machine learning models;

(2) Exploring explainability techniques to make automated refactoring recommendations more transparent and understandable. The "black-box" nature of machine learning models can hinder the practical adoption of recommendations. Increasing transparency also facilitates adoption, as users better understand why a recommendation is made and which code features influenced the suggestion;

(3) Investigating hybrid approaches that combine machine learning algorithms with diverse types of features aimed at capturing the varied nature of user preferences, coding styles, and contextual factors. It is worth noting that a large number of studies in this systematic review rely primarily on object-oriented metrics; however, this reliance does not guaranty an adequate representation of such diversity;

(4) Enhancing the granularity of recommendations to identify all components involved in a recommendation (such as classes and methods). This systematic review showed that refactorings do not always include all the necessary code elements. For example, [A04] approach indicates the type of refactoring to be applied, but not precisely where it should be applied;

(5) Incorporating user feedback mechanisms to iteratively improve the quality and

relevance of recommendations. Feedback is a crucial aspect of understanding user preferences and improving the machine learning model, allowing for better recommendations over time (PANTIUCHINA et al., 2021).

3.7 Final Considerations

This chapter presents a systematic mapping of how Machine Learning has been used for Refactoring recommendation approaches. The final set of 27 papers, grouped into 22 approaches, was reviewed and analyzed to answer the three main research questions and sub-questions defined.

Chapter 4
THE W3B CRITERIA

4.1 Initial Considerations

Chapter 3 presented a systematic review examining how machine learning has been applied to refactoring recommendations. This review raises an important question: what constitutes a *complete* recommendation? In this chapter, we outline the key characteristics that, in our view, define a complete recommendation.

4.2 Complete Recommendation

The systematic review conducted in Chapter 3 revealed a significant gap in the formulation stage of refactoring recommendations, particularly regarding how the recommendations are presented to end users. While many state-of-the-art approaches focus on optimizing performance metrics such as precision, recall, and F-measure, our work emphasizes a different perspective: examining the components that constitute the refactoring recommendations. This distinction highlights a complementary focus of the already existing approaches.

We argue that a complete recommendation is one that provides developers with the relevant information they need to make a decision. In this work, we use the term **complete** to describe recommendations that specify: i) the name of the refactoring to be applied, ii) the affected software components (class, method, fragment, etc); iii) the rationales behind the recommendation; and iv) the potential benefits. Such completeness

enhances trust and facilitates informed decision-making by the user. To enable a systematic assessment of completeness, we propose a classification criterion named W3B (**W**hich, **W**here, **W**hy and **B**enefits), explained below.

4.2.1 W3B (Which, Where, Why and Benefits)

The W3B criterion arises directly from the systematic review, specifically based on the analysis of all articles in our final set and the format in which they deliver recommendations to end users. This criterion provides a structured way to assess the completeness of the refactoring recommendations by examining whether they specify **which** refactoring should be applied, **where** it should be applied, **why** it was recommended, and the **benefits** it may provide. All components of the criterion are detailed below.

WHICH: This involves a precise identification of **which** refactoring(s) should be applied, including the specific refactoring name. For example: Extract Method, Rename Method or Move Method. Some approaches recommend applying a single refactoring at a time (CUI et al., 2023; PALIT et al., 2023; ALOMAR et al., 2023), while others suggest an ordered sequence of refactorings (NYAMAWE, 2022; REBAI et al., 2020; ALIZADEH et al., 2019).

WHERE: This involves a clear identification of **where** the refactoring should be applied, identifying the parts of the source code involved, such as, methods, fields, classes, parameters, among others. Table 9 provides examples of *Where* components for three refactorings. For *Extract Method*, it is necessary to identify both the original method and the specific code fragment to be extracted, this refactoring puts the new extracted method in the same class, so the target class is the same of the origin class. In the case of a *Move Method*, the recommendation should clearly indicate the source and destination classes, as well as the method to be moved. The granularity of the affected components may vary depending on the type of refactoring, ranging from small code fragments to entire classes or packages. Accurately specifying the target elements is crucial to ensure correct application of the refactoring recommended. Thus, the *where* element is inherently complex, as its definition depends on the type of refactoring being recommended.

Table 9 – Examples of *Where* in W3B

Refactoring	**Where – Software Components**
Extract Method	1) *original Method* , method that will undergo the extraction 2) Fragment of the *original Method* to be extracted 3) The *name* of the new method after the extraction
Move Method	1) *analyzed Method* 2) *original Class* of the *analyzed Method* 3) *Target Class*, new class that will contain the *analyzed Method*
Pull Up Method	1) *duplicated Method* present in two or more subclasses 2) *Subclasses 1* containing the *duplicated Method* 3) *Subclasses 2* containing the *duplicated Method* 4) *Target Superclass*, where the Method will be moved
Encapsulate Field	1) *field* to be encapsulated 2) *getter Method* created for field access 3) *setter Method* created for field modification

WHY: This involves a clear explanation of *why* a particular refactoring was recommended. A simple way to think about this is asking "WHY was the refactoring *R* recommended?". The answer must elucidate the rationale behind the decision. From a developer's perspective, it is crucial to clarify why such a refactoring was recommended. Consequently, a recommendation should convey this information explicitly, such as: *This refactoring is being recommended because this piece of code (method/class) has a cohesion around X and a coupling with n other classes*. A clear rationale enables developers to assess whether to accept or disregard the recommendation based on their understanding of the project. In our context, providing a transparent explanation is essential for users to understand and trust the recommendation of the Machine Learning model. This objective can be achieved through the application of Explainable Artificial Intelligence (XAI) techniques (DAS; RAD, 2020) that offer feature-based explanations, highlighting the most influential features for the recommendation decision.

BENEFITS: This outlines the benefits expected from applying the refactoring, helping the developer anticipate the improvements in the system's maintainability, performance, readability, or other quality attributes. A clear articulation of the benefits provides additional motivation for accepting the recommendation.

To illustrate the application of the *W3B* criteria in a real-world scenario, we focus on the Extract Method refactoring and the Java method called *afterTextChanged*. To understand the method, we describe the context of the method.

The *afterTextChanged* method monitors the text entered in an EditText and ensures that it behaves as a single-line field when appropriate. If no newline characters are detected and the input is not intended to be multi-line, the method disables the multi-line input flag while preserving the cursor position, preventing unexpected behavior. In this way, it enforces single-line input unless the user explicitly enters a line break, maintaining the intended input behavior dynamically.

```
1  public void afterTextChanged(Editable s) {
2      if (mPossiblyNotMultiline) {
3          boolean found = false;
4          for (int i = s.length() - 1; i >= 0; --i) {
5              if (s.charAt(i) == '\n') {
6                  found = true;
7                  break;
8              }
9          }
10         if (!found) {
11             mMultiline = false;
12             int pos = mEditText.getSelectionStart();
13             mEditText.setInputType(mEditText.getInputType()
14                     & (~InputType.TYPE_TEXT_FLAG_MULTI_LINE));
15             mEditText.setSelection(pos);
16         }
17     }
18 }
```

Listing 4.1 – Example of W3B

Each method may have several candidate fragments. For our example, the following code fragment was chosen as the best candidate.

```
1  mMultiline = false;
2  int pos = mEditText.getSelectionStart();
3  mEditText.setInputType(mEditText.getInputType() & (~InputType.TYPE_TEXT_FLAG_MULTI_LINE
       ));
4  mEditText.setSelection(pos);
```

Listing 4.2 – Candidate Source Code fragment

So the recommendation in accordance with the **W3B** criteria would be as shown below:

(1) Which: Extract Method refactoring

(2) Where: i) the *afterTextChanged* method for extraction 4.1; ii) the fragment to be extracted 4.2; iii) *updateMultilineStatus*, the name of the created method.

(3) Why: i) The high *Response For a Class* (RFC) value of 6 suggests that the method is potentially complex, as it could trigger multiple methods; ii) The *maximum depth of nested code blocks* is 3, which further contributes to the method's complexity; iii) The method has 17 lines of code and 5 assignment statements, indicating that it is relatively large and contains multiple operations.

(4) Benefits: i) encapsulates the logic that modifies the mEditText properties into a separate method, which abstracts away implementation details; ii) enhances the readability of the main method by focusing only on its high-level structure; iii) enables potential reuse of the extracted logic in other parts of the project.

4.2.2 Systematic Review and the W3B Criteria

We apply W3B to assess the completeness of the approaches in the systematic review (SR) which considers a total of 22 approaches (ID [A01] to [A22]) conducted in Chapter 3. The result of applying the criteria can be observed in Table 10. The first column presents the ID of the approach, the second column indicates the paper ID and columns three to six correspond to the W3B elements: *which*, *where*, *why*, and *benefits*. The benefits were identified from the papers and incorporated into the Table, even though they were not explicitly included in the recommendation. The last column indicates whether the approach satisfies all four elements, marking those that do.

Table 10 – The Final Set of Primary Studies

ID	Paper	**Which** refactoring is recommended	**Where**	**Why**	**Benefits**	**W3B**
[A01]	[S01]	Extract Method	Complete	Undefined	Solving Duplicated Code, Feature Envy and Long Method	✗
[A02]	[S02]	Extract Method	Incomplete	Undefined	Undefined	✗

continued on next page

Table 10 – continuation from previous page

ID	Paper	**Which** refactoring is recommended	**Where**	**Why**	**Benefits**	**W3B**
[A03]	[S03]	Extract Method	Complete	Pop-up notification	Solving Code clone smell	✓
[A04]	[S04] [S12] [S19]	Extract Interface, Extract Method, Extract Superclass, Inline Method, Move And Rename Class, Move Attribute, Move Class, Move Method, Pull Up Attribute, Pull Up Method, Push Down Attribute, Push Down Method, Rename Class, Rename Method,	Undefined	Undefined	Adapting the system for new requirements; improve code cohesion and keep the conformity to OOP principles	✗
[A05]	[S05] [S17] [S20]	Extract Class, Extract SubClass, Extract SuperClass, Extract Method, Move Method/Field, PullUp Field, PullUp Method, PushDown Field/Method, Encapsulate Field, Increase Field Security, Decrease Field Security, Increase Method Security, Decrease Method Security	Complete	Graphical charts and tables	Improving quality attributes QMOOD, in terms of Reusability, Flexibility, Understandability, Functionality, Extendibility, Effectiveness.	✓
[A06]	[S04]	Move Method	Complete	Undefined	Solving Feature Envy	✗
[A07]	[S07]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability.	✗
[A08]	[S08]	Move Operation, Move Attribute, Extract Class, Extract Associated Class, Extract Subclass, Extract Superclass, Pull Up Operation, Pull Up Attribute, Push Down Operation, Push Down Attribute, Rename Class, Rename Operation, Rename Attribute.	Undefined	Undefined	Identifying Functional decomposition in UML diagrams.	✗
[A09]	[S09]	Remove relations, Remove classes and Add classes	Complete	Undefined	Solving God Class, Data Class, Feature Envy, Complex Class, Spaghetti Class, and Speculative Generality.	✗
[A10]	[S10]	Extract Method	Incomplete	Undefined	Undefined	✗

continued on next page

Table 10 – continuation from previous page

ID	Paper	**Which** refactoring is recommended	**Where**	**Why**	**Benefits**	**W3B**
[A11]	[S11]	Extract Class, Extract Subclass, Extract Super-class, Extract Interface, Move Class, Rename Class, Move and Rename Class, Extract Method, Inline Method, Move Method, Pull Up, Push Down Method, Rename Method, Extract And Move Method, Extract Variable, Inline Variable, Rename Variable.	Incomplete	Undefined	Undefined	✗
[A12]	[S13]	Move Method	Complete	Undefined	Solving Feature Envy smell; reducing the coupling between classes	✗
[A13]	[S14]	Extract Method, Pullup Method, substitution Algorithm, Replace Temp with Query, Introduce parameter Object, Preserve the whole object, Replace method with method object, decomposition objects	Incomplete	Undefined	Duplicate code and Long Method code smells	✗
[A14]	[S15]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability;	✗
[A15]	[S16]	Move Method, Pull up Method, Extract Method	Incomplete	Undefined	Solving code clone type I, II and III	✗
[A16]	[S18] [S21]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability	✗
[A17]	[S22]	Extract Method	Complete	Undefined	Solving Code clone smell	✗
[A18]	[S23]	Move Method	Complete	Undefined	Solving Feature Envy code smell	✗
[A19]	[S24]	Extract Method	Complete	Undefined	Long Method smell	✗
[A20]	[S25]	Extract Method	Complete	Undefined	Improving the software maintainability and source code readability	✗
[A21]	[S26]	Extract Method	Incomplete	Undefined	Improving the software maintainability	✗

continued on next page

Table 10 – continuation from previous page

ID	Paper	**Which** refactoring is recommended	**Where**	**Why**	**Benefits**	**W3B**
[A22]	[S27]	Undefined by the authors	Incomplete	Undefined	Improve the software maintainability	✗

To better understand the distribution of these elements across the evaluated approaches, Table 11 and Figure 5 provide an overview of the counts and percentages of approaches satisfying each element. The *Which* and *Benefits* elements are the most frequently satisfied, with over 80% of approaches clearly indicating the recommended refactoring and the expected benefits. In contrast, the *Where* element is satisfied in less than half of the cases (approximately 41%), suggesting that many studies do not specify precisely the location or context *where* the refactoring should be applied.

Table 11 – Quantitative Analysis of W3B Elements Satisfaction

Element	Quantity	Percentage (%)
Which	18	81.8
Where	9	40.9
Why	2	9.1
Benefits	19	86.4

Moreover, the *Why* element is the most neglected, explicitly present in only about 9% of the approaches. This lack of explanation for the rationale behind refactoring recommendations reveals a significant gap, which may hinder users' understanding of the motivation and the decision-making process in automated refactoring approaches.

Further examination of Table 10 shows that approaches [A01], [A03], [A05], [A06], [A09], [A12], [A17], [A18], [A19], and [A20] have successfully identified **Which** refactorings should be applied and provided comprehensive insights into the precise **Where** these refactorings should be applied. This means that 45% of the approaches met at least these two elements of the criterion, with most of them recommending only a single refactoring.

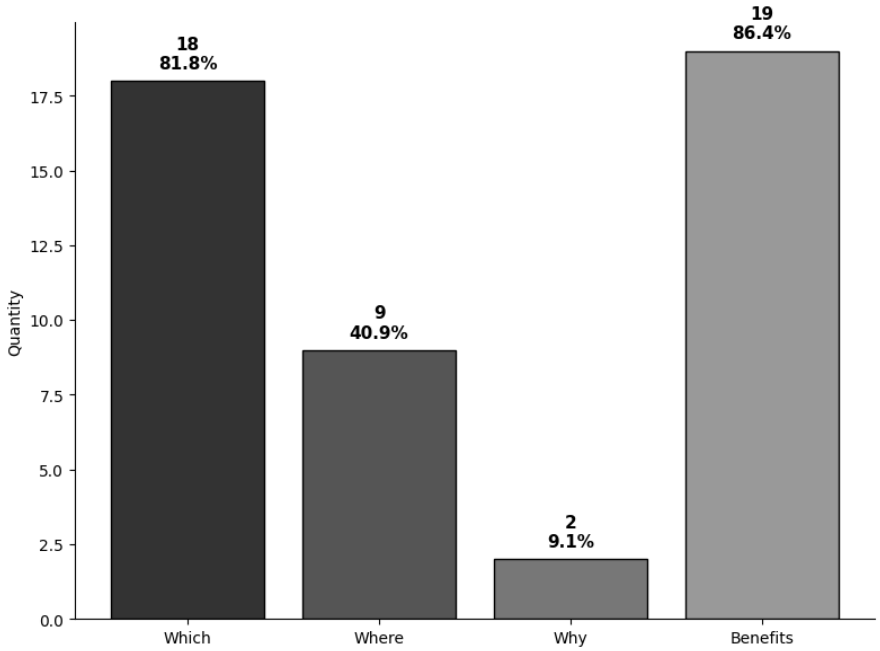


Figure 5 – Satisfaction of W3B Elements in Approaches

For the **Where** element, the terms *Complete* or *Incomplete* depend on the type of refactoring recommended. In the case of the approaches [A04] and [A08] do not specify the exact source code component where the refactoring should be applied, leaving this task to the developer. It is interesting to mention that in the approach [A04] (papers [S04], [S12] and [S19]), the authors acknowledge this limitation and suggest using an external tool to fill this gap.

Approaches [A07], [A14], [A16], and [A22] do not identify the **Which** element at all. Instead, they focus on the identification and recommendation of the component (method or class) that must be refactored, providing a high-level recommendation. However, the absence of the **Which** element directly impacts the completeness of **Where**, since the sub-elements of **Where** depend on the specific type of refactoring being recommended.

Regarding **Why**, only approaches [A03] and [A05] explicitly explain the rationale behind their recommendations. Approach [A03] displays a pop-up notification at the bottom of the screen, alerting the developer to an opportunity to apply *Extract Method*

and explaining why it is advisable. Approach [A05] uses interactive tables and charts along with detailed analysis to justify its recommendations. This finding indicates that almost 90% of the papers do not provide any explanation for the rationale behind a recommendation.

As for **Benefits**, 11 approaches ([A01], [A03], [A06], [A08], [A09], [A12], [A13], [A15], [A17], [A18], and [A19]) aim to resolve a code smell. The relationship between code smells and refactorings is well-established in the software engineering research community, with several secondary studies highlighting which refactorings address which smells. Therefore, it is unsurprising to see this benefit in the final set of articles. Another stated benefit is "improving software maintainability," mentioned in 6 approaches ([A07], [A14], [A16], [A20], [A21], [A22]). However, this is a broad and generic goal, as the authors do not specify which aspects of maintainability they aim to improve.

Overall, only approaches [A03] and [A05] satisfy all four elements of the W3B framework, meaning that more than 90% of the reviewed approaches cannot be considered complete according to this criterion. These results provide evidence of a significant gap in the field of refactoring recommendation and, at the same time, highlight opportunities for future research, particularly in developing recommendations that are *complete* for developers.

As a proposed evaluation criterion, W3B can also be adopted by other researchers and practitioners to design, document, and assess refactoring recommendation systems in a more transparent and user-centered manner.

4.3 Discussion

The analysis presented in Table 10 shows that the *Which* and *Where* elements of the *W3B* criterion are addressed by approximately 45% of the evaluated approaches. However, a closer examination of the individual elements reveals that *Why* and *Benefits* are significantly neglected. Specifically, only about 9% of the approaches provide clear justifications for applying the refactoring (*Why*), while the *Benefits* element is present in less than 20% of the cases.

The absence of the *Why* and *Benefits* elements may have practical implications

for the adoption process of refactorings. The lack of explicit justifications (*Why*) tends to reduce developers' confidence in the recommendations, since understanding the motivations behind the recommendations is fundamental for informed decision-making (NYAMAWE, 2022) (ALOMAR et al., 2023) (REBAI et al., 2020) (CUI et al., 2023). Similarly, not directly showing the benefits of refactoring in the recommendation (*Benefits*) compromises the perception of added value, making it difficult to evaluate the cost-benefit ratio and, consequently, hindering effective adoption in real-world contexts.

Furthermore, the low coverage of the *Why* element can be attributed to both the methodological limitations inherent in the evaluated approaches and the complexity of formalizing automatic explanations that are comprehensible to developers (See Table 8 in Chapter 3). It may also reflect the authors' predominant focus on the technical aspects of recommendations. Thus, the incorporation of explainability techniques (Explainable AI) stands out as a way to improve the *Why* element, providing transparent and well-founded justifications.

4.4 Final Considerations

This chapter presents a comprehensive analysis of refactoring recommendation approaches using the W3B criterion, highlighting key strengths and significant gaps in their completeness. While many approaches adequately address the identification of what refactorings to apply (*Which*) and where to apply them (*Where*), the elements related to the underlying rationale (*Why*) and the expected advantages (*Benefits*) remain largely underexplored. This gap may undermine developer confidence and hinder the practical adoption of these recommendations.

Despite the limitations inherent in applying the W3B criteria, particularly considering the diversity of research goals and contexts, it remains a valuable criterion. Furthermore, the W3B can also be adopted by other researchers and practitioners to design, document, and assess refactoring recommendation systems in a more transparent and user-centered manner.

Chapter 5
A Consensual Strategy for Explainability

5.1 Initial Considerations

Understanding the decision-making process of black-box models has become increasingly relevant in contexts where trust and interpretability are essential, such as in refactoring recommendation systems. Although several explanation techniques (e.g., SHAP, LIME, Anchor) have been proposed, they often produce divergent results, leading to inconsistency and reduced confidence in automated decisions. To address this, we propose a consensual strategy that combines multiple explainers to generate unified, more robust explanations. This strategy was implemented as a publicly available prototype and evaluated through an empirical study.

5.2 Context and Motivation

The goal of this work is to formulate complete Extract Method refactoring recommendations. We believe that such recommendations could benefit from clear explanations, as they assist developers in understanding the rationale behind the suggested changes. This not only increases trust in automated tools but also reduces the likelihood of developers blindly accepting or rejecting recommendations. Transparent explanations help developers assess whether a proposed refactoring aligns with their design intentions, coding style, or architectural goals, ultimately supporting better-informed decisions and

improving software quality (ARMIJO et al., 2024; PANTIUCHINA et al., 2021; CUI et al., 2023; ALOMAR et al., 2023; CUNHA et al., 2020; KAUR; RATTAN, 2023).

The pre-trained CodeBERT model demonstrated outstanding performance in predicting tasks. However, it functions as a *black box* where the internal decision-making processes remain opaque to end users. To overcome this limitation, the field of Explainable Artificial Intelligence (XAI) has emerged, aiming to increase the transparency of ML models by providing understandable justifications for their predictions (GUNNING et al., 2019; BOMMER et al., 2024). Among the most widely adopted explanation techniques, commonly referred to as *explainers*, are LIME (Local Interpretable Model-agnostic Explanations) (RIBEIRO et al., 2016), SHAP (SHapley Additive exPlanations) (LUNDBERG; LEE, 2017b), and Anchors (RIBEIRO et al., 2018). These methods have been applied in various domains to help users understand and trust automated decisions.

However, during the development of this approach, we encountered a critical issue: the *disagreement among existing explainers*. It occurs when different explanation techniques often yield inconsistent justifications for the same model output, leading to contradictory feature importance rankings and reducing user confidence in the explanations themselves (KRISHNA et al., 2024; ROY et al., 2022a; PIRIE et al., 2023). This lack of consistency made it difficult to select a single explainer and raised broader concerns about the reliability and interpretability of XAI methods.

Several studies have attempted to address this issue through different strategies, such as functional decomposition (LABERGE et al., 2024), alignment of explainer outputs (PIRIE et al., 2023), regularization with loss terms (SCHWARZSCHILD et al., 2023), and feature intersection constraints (BANEGAS-LUNA et al., 2023). While promising, many of these approaches are experimental or require users to understand intricate technical mechanisms. Furthermore, they often lack flexibility for customization and adaptation to specific application domains.

To overcome these limitations, we developed a new solution: a *consensual explainer* capable of aggregating outputs from multiple explanation techniques into a unified, more stable explanation. The proposed strategy mitigates inconsistency by identifying agreement patterns among diverse methods. It was designed as a standalone, configurable tool, allowing users to select internal explainers, assign weights, define the

number of top-k features, and adjust the strictness of feature intersections according to their needs.

In addition to its independent use, a streamlined version with fixed parameters was integrated as a component of our refactoring recommender system. This integration enables each recommendation to be accompanied by a robust, domain-tailored explanation derived from the consensus across multiple methods, thereby improving transparency and trust in the automated refactoring process.

5.3 Detailing the Consensual Explainer Process

Figure 6 shows an overview of the Consensual Explainer Module. Part (a) of Figure 6 illustrates the required inputs for using the Consensus Module. To use our module, the user must provide three elements: a trained machine learning model, the associated dataset, and the instance to be explained. The intended user is someone seeking a unified explanation for why a machine learning model produced a specific output for a given instance. Often, such users have previously explored individual explainers such as SHAP, LIME, or Anchors, but encountered conflicting results.

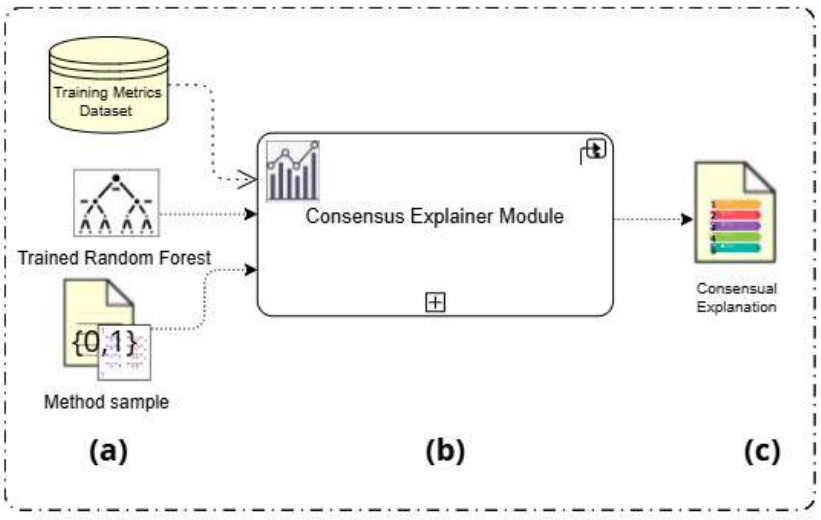


Figure 6 – Consensual Explainer Module

Part (b) presents the internal structure of the Consensual Module, the core component of our approach. It comprises six processing steps responsible for aggregating and reconciling the outputs from each embedded explainer to generate a single, unified explanation. By default, our implementation includes SHAP (LUNDBERG; LEE, 2017a), LIME (RIBEIRO et al., 2016), and ANCHOR (RIBEIRO et al., 2018), selected for their complementary strengths: SHAP delivers theoretically grounded, globally consistent attributions; LIME offers locally faithful, model-agnostic explanations; and ANCHOR provides rule-based interpretations with high precision. This combination enables our module to capture diverse interpretability perspectives, local and rule-based, thereby improving the robustness and credibility of the consensus. The architecture is flexible, allowing for the integration of new explainers that generate local explanations based on feature relevance, such as Integrated Gradients (SUNDARARAJAN et al., 2017) and RISE (PETSIUK et al., 2018).

Part (c) shows the final output of the module: a consensus explanation presented as a ranked list of relevant features. Each feature is accompanied by a weight and an agreement index, indicating the level of support it received from the internal explainers. Thus, the user receives a consistent and interpretable rationale behind the model's decision, addressing the limitations of relying on a single explainer.

5.4 Internal structure of the Consensual Module

Figure 7 presents the details of our approach for building the consensual explainer, organized into six sequential steps labeled with letters. Steps A, B, and C address the *Treatment of the Explainer*, Steps D and E focus on the *Feature Agreement*, and Step F concludes the process.

Step A. Setting Up and Running the Explainer. The step begins with the instantiation of the selected explainers E_n. To do so, it requires two key artifacts: the *trained machine learning* model and the *Training Metrics Dataset*. Once instantiated, the explainer is executed using the *instance*, referring to the specific *Sample Method* for which an explanation is to be generated. The output of this process is a feature-based explanation, presented as a ranked list of features, where each feature is assigned a score that reflects its relative contribution to the model prediction for that instance.

Since each explainer (represented as E_1, E_2, and E_n) produces outputs in varying formats, we define a *unified internal representation* to ensure consistency across all explainers. Each explanation based on features is transformed into this common structure, which includes: (i) the name of the feature, (ii) the feature value; (iii) the contribution weight assigned by the explainer, (iv) the feature's position in the explainer ranking; and (v) the range of values (when the explainer provides it). This representation ensures a consistent and structured view of heterogeneous explainer outputs. By standardizing the format of feature-based explanations, the approach facilitates direct comparison, aggregation, and the processing of results.

Step B. Filtering features by sign. In this step, two considerations are assumed: i) We consider only positive samples. In the context of refactoring recommendations, our focus is on samples where the Recommender predicts the value of 1, i.e., the method should be refactored. Consequently, all instances analyzed in this work correspond to predictions equal to 1; and ii) Each explanation contains features with both positive and negative contributions. For our proposal, we filter the explanation of each explainer E_n, using the Sign Agreement (SA) metric (KRISHNA et al., 2024). The Sign Filter is then applied to systematically remove features that negatively affect the outcome. Specifically, those that push the prediction away from recommending refactoring.

Step C. Sorting and Tagging Features. The first task in this step is to sort the filtered features based on their contribution weights in ascending order. If the explainer does not provide explicit weights, the features are ordered by their original ranking positions. Each explainer E_n assigns a weight that reflects how strongly a feature influences the prediction. To enable comparisons, these weights are normalized to a [0, 1] scale.

Next, each feature is tagged with its original ranking position to preserve the importance assigned by its explainer. This positional information is essential in subsequent steps for identifying and aligning common features across multiple explainers. After completing this step, the process returns to *Step A* if there are additional explainers to run; otherwise, it advances to the next steps.

To illustrate the following steps of the process, we present a partial analysis that considers the top three features ranked by each explainer. Listings 5.1, 5.2, and 5.3 show the ranked features selected by the Anchors, SHAP, and LIME explainers, respectively,

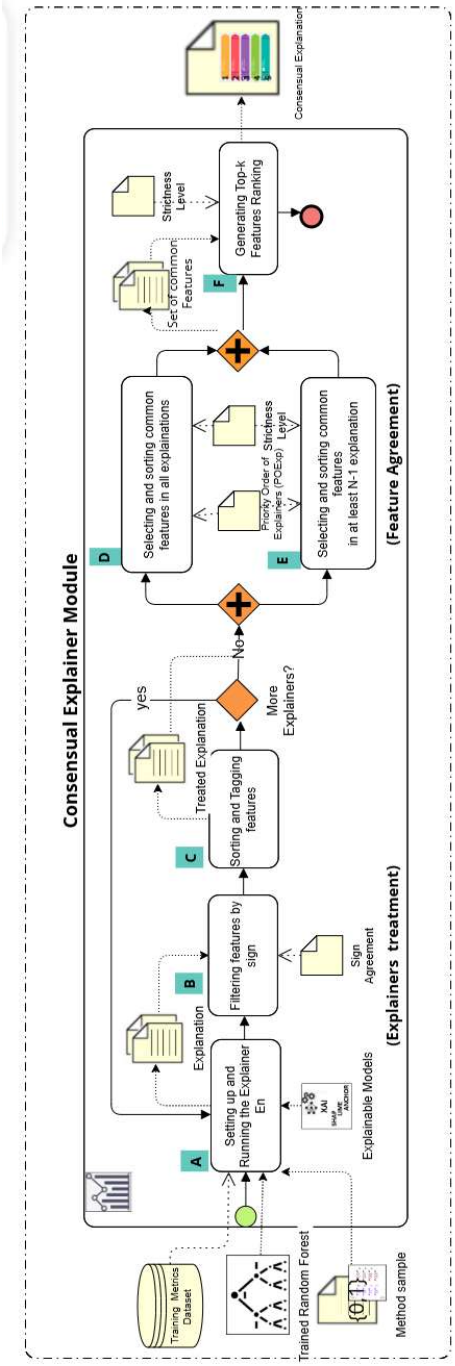


Figure 7 – Process of building the Consensual Explainer Module

after being processed through Steps A to C.

```
"features": [
    {
      "feature_name": "methodRfc",
      "feature_value": 25,
      "feature_weight": 0.27343263587118105,
      "feature_ranges": "methodRfc > 10.00",
      "feature_rank": 1
    },
    {
      "feature_name": "methodUniqueWordsQty",
      "feature_value": 65,
      "feature_weight": 0.3498896924350002,
      "feature_ranges": "methodUniqueWordsQty > 13.00",
      "feature_rank": 2
    },
    {
      "feature_name": "methodStringLiteralsQty",
      "feature_value": 1,
      "feature_weight": 0.1542290616328296,
      "feature_ranges": "methodStringLiteralsQty <= 2.00",
      "feature_rank": 3
    }
    ...
    ...
  ]
```

Listing 5.1 – Part of the result of the Anchors explanation

```
"features": [
    {
      "feature_name": "methodRfc",
      "feature_value": 25,
      "feature_ranges": null,
      "feature_weight": 0.2148079127977562,
      "feature_rank": 1
    },
    {
      "feature_name": "methodLoc",
      "feature_value": 44,
      "feature_ranges": null,
      "feature_weight": 0.16953078666190213,
      "feature_rank": 2
    },
    {
      "feature_name": "methodUniqueWordsQty",
      "feature_value": 65,
      "feature_ranges": null,
      "feature_weight": 0.1325822046136422,
```

```
      "feature_rank": 3
    }
    ...
    ...
    ]
```

Listing 5.2 – Part of the result of the Shap explanation

```
"features": [
    {
      "feature_name": "methodRfc",
      "feature_value": 25,
      "feature_ranges": "methodRfc > 10.00",
      "feature_rank": 1,
      "feature_weight": 0.28438423150512243
    },
    {
      "feature_name": "methodLoc",
      "feature_value": 44,
      "feature_ranges": "methodLoc > 24.00",
      "feature_rank": 2,
      "feature_weight": 0.1965167976903637
    },
    {
      "feature_name": "methodUniqueWordsQty",
      "feature_value": 65,
      "feature_ranges": "methodUniqueWordsQty > 26.00",
      "feature_rank": 3,
      "feature_weight": 0.18798758737038923
    }
    ...
    ...
    ]
```

Listing 5.3 – Part of the result of the Lime explanation

Step D. Selecting and sorting common features in all explanations.

The main objective of *Step D* and *Step E*, considered as feature agreement grouping, is to generate ordered sets of features that are common across the *n* explanations obtained in the previous steps. The number of sets generated depends on the desired *strictness level*, which controls how much agreement is required among the explanations.

The strictness level reflects the trade-off between explanation consensus and feature inclusiveness. For instance, if high agreement is required, only features present in all *n* explanations will be included. Conversely, if more flexibility is acceptable, features

shared by *n-1* or even *n-2* explanations may be considered. The choice of level typically depends on the domain, the specific goal of the explanation, and the engineer's expertise. To enable this customization, we define a user-configurable parameter named *strictness level*, with three possible values:

- **High**: Includes only features that are common across all *n* explanations, resulting in a single ordered set. This strict configuration reflects the highest level of agreement.
- **Medium**: Considers features that appear in at least *n–1* of the *n* explanations, producing two ordered sets. This setting balances strictness and flexibility, allowing for slight variations while still maintaining significant consensus.
- **Low**: Incorporates features found in at least *n–2* of the *n* explanations, generating three ordered sets. This more relaxed threshold captures broader agreement and is useful when exploring diverse explanatory patterns.

For each level, the algorithm executes a series of iterations. In each iteration, it performs two actions: (I) Identifies features shared by a specific number of explanations; (II) Sorts those features to produce an ordered set. As an example, with strictness set to low and $n = 5$, the algorithm will generate three sets by identifying features common to 5, 4, and 3 explanations, respectively. Among the resulting sets, the first is the most informative, as it contains features consistently present across all explanations.

Step E. Selecting and sorting common features in at least N-1 explanation.

The sorting step relies on an artifact called Priority Order of Explainers (POExp), shown in Table 12. This artifact defines the relative priority of each explainer when resolving conflicts in feature rankings. For example, if a feature appears in all explanations but in different positions, the POExp is used to determine its final rank within the set.

By default, the POExp prioritizes Anchors over other explainers, such as LIME, which has the lowest priority. Anchors are favored because they produce rule-based explanations known as "anchors", which guarantee prediction stability and enhance the interpretability and computational scalability of the explanation process (RIBEIRO et al., 2018).

Priority Order of Explainers and weights (default)		
Priority	Explainer	weight
1st	Anchors	E1_weight, default value: 3
2nd	SHAP	E2_weight, default value: 2
3rd	LIME	E3_weight, default value: 1

Table 12 – Priority Order of Explainers (POExp) Artifact

Step F. Generating Top-k Features Ranking.

In this final step, we receive an input *k* that defines how many features we will select for the final ranking and two sets of common features from previous steps, by default *k = 5*. We get all the features from the first set of common features (features that are common in all Explainers, selected in *Step D*) and fill the top-k features ranking, within the limit of the first *k* features. If the first set of features has fewer than *k* features, we complete the ranking with the second set of features (features that are common in at least two Explainers, selected in *Step E*.

Additionally, in this step we calculate two indices called *feature weight* and *feature agreement index*. These two indices complement each other. Thus, the feature weight allows for comparison with the ranking of other explainers, while the agreement index shows the consensus among them regarding the feature.

1. The average weight assigned to each feature is determined by the importance of each explainer in the POExp table. The Explainers are denoted by E1, E2, E3, ..., En, and their respective relative weights of the explained represented as E1_weight, E2_weight,...,En_weight. Below is an example of how the weighted average is calculated, based on the POExp priorities from the Table 12. In the formula, *fn_weight* represent the original weight of the feature by the explainer.

$$\text{feature\_weight} = \frac{\sum_{i=1}^{n} E_i\_weight \cdot f_i\_weight}{\sum_{i=1}^{n} E_i\_weight} \tag{5.1}$$

2. The agreement index represents the percentage of explainers who agree with the selected feature. To this end, the weight specified in the Table 12 is used to calculate

a weighted index. In the formula, *E1_bool* (values 1 or 0) indicates whether the explainer considers the feature in its ranking.

$$\text{feature\_agreement\_index} = \frac{\sum_{i=1}^{n} E_i\_weight \cdot E_i\_bool}{\sum_{i=1}^{n} E_i\_weight} \tag{5.2}$$

To illustrate how the metrics are calculated and highlight their importance in generating the final ranking, we present the computation of the values shown in Listings 5.1, 5.2, and 5.3. An agreement index of 1.0 indicates that the corresponding feature was consistently ranked by all three explainability models, reflecting full inter-model consensus. Conversely, the Method LOC feature exhibits an agreement index of 0.50, as it was included in the rankings of only two out of the three models.

- Feature: **methodRfc**

$$\begin{aligned}\text{feature\_weight(methodRfc)} &= \frac{3 \times 0.2734 + 2 \times 0.2148 + 1 \times 0.2844}{3 + 2 + 1} \\ &= \frac{0.8202 + 0.4296 + 0.2844}{6} \\ &= \frac{1.5342}{6} \\ &= \boxed{0.2557}\end{aligned}$$

$$\text{agreement\_index(methodRfc)} = \frac{6}{6} = 1.0$$

- Feature: **methodUniqueWordsQty**

feature_weight (methodUniqueWordsQty) = 0.2505
agreement_index (methodUniqueWordsQty) = 1.0

- Feature: **methodAssignmentsQty**

feature_weight (methodAssignmentsQty) = 0.0717
agreement_index (methodAssignmentsQty) = 1.0

- Feature: **methodLoc**

$$\begin{aligned}\text{feature\_weight}(\text{methodLoc}) &= \frac{2 \times 0.1695 + 1 \times 0.1965}{2+1} \\ &= \frac{0.3391 + 0.1965}{3} \\ &= \frac{0.5356}{3} \\ &= \boxed{0.1785}\end{aligned}$$

$$\text{agreement\_index}(\text{methodLoc}) = \frac{3}{6} \approx 0.5$$

Listing 5.4 presents the ranked features according to our consensual strategy. Each entry combines the feature name, its normalized importance weight (expressed as a percentage), and its agreement index, representing the proportion of explainers (among SHAP, LIME, and Anchors) that included the feature in their top rankings. For instance, *methodRfc* achieved the highest relative importance (25.6%) and was unanimously selected by all explainers (100% agreement), while *methodLoc* was included by only two out of the three models, resulting in a lower agreement index (66.6%). This output reflects how the aggregation process favors features that are not only locally important but also consistently highlighted across different explanation methods.

```
"our_approach": [
    "methodRfc (25.6% - 100.0%)",
    "methodUniqueWordsQty (25.0% - 100.0%)",
    "methodAssignmentsQty (7.2% - 100.0%)",
    "methodLoc (17.9% - 50.%)".
    "methodStringLiteralsQty (10.6% - 83.3%)
  ]
```

Listing 5.4 – Top-5 Ranked Features According to Our Model

5.5 An Empirical Analysis

The goal of this empirical analysis is to ***compare the agreement level between our explanation and the baseline Explainable Boosting Machine (EBM), LIME, SHAP and Anchors, using the metrics Feature Agreement (FA) and Rank Agreement (RA) in the context of an Extract Method Recomendation.*** (KRISHNA et al., 2023) (KRISHNA et al., 2024).

Since our consensus approach integrates SHAP, LIME, and Anchors, we expect it to achieve higher agreement levels than any individual explainer. By including these baseline methods in our comparison, we aim to verify whether the consensus explanation truly enhances consistency rather than simply replicating the behavior of a single method.

To support this comparison, we compute the average FA and RA metrics between pairs of explainers across all instances. For example, when comparing two explainers, E1 and E2, we apply both to the same set of n labeled instances, generating n explanations per explainer. For each instance, we compute the agreement metric (FA or RA) between the corresponding explanations. This yields n agreement scores, which we then average to obtain a single value representing the overall agreement between E1 and E2. This evaluation strategy follows the methodology proposed by (KRISHNA et al., 2024).

5.5.1 The Datasets used

The source dataset used in this study was originally compiled by Aniche et al. (ANICHE et al., 2020). It includes process, class, and method-level metrics associated with different types of refactorings. For the purpose of this work, we focus exclusively on the Extract Method refactorings, belonging to projects publicly available on GitHub, and developed in Java.

We select 249911 samples from the source to construct two specialized datasets: one of 150000 called *Training Metrics Dataset* for training the Random Forest model and the other with 99911 called *Evaluating Metrics Dataset*. The datasets are composed of 23 method-level metrics that best capture the structural and behavioral complexity of object-oriented methods. These include: Lines of Code (LOC), Coupling Between Objects (CBO), Cyclomatic Complexity (CC), etc. which are commonly used indicators in the context of refactoring analysis.

The dataset*Training Metrics Dataset* is composed of 150000 instances; 75000 positive samples and 75000 negative samples. For the *Evaluating Metrics Dataset*, we considered only positive instances. Since our primary goal is to analyze the explanations behind positive refactoring recommendations, we restricted the evaluation set to positive instances only.

5.5.1.1 Supervised and Unsupervised Learning Algorithms used

For the training stage, we used the Random Forest motivated by its robustness, interpretability, and effectiveness in handling high-dimensional datasets that have been widely reported as described in the systematic review conducted (CUI et al., 2023) (PALIT et al., 2023) (ALOMAR et al., 2023) (XU et al., 2017a) (IMAZATO et al., 2017). Random Forest models are particularly suitable for explainability purposes, as they are naturally compatible with *post-hoc* and *model-agnostic* explanation techniques.

For the evaluation stage, we use the *Evaluating Metrics Dataset* with the K-means clustering algorithm to uncover potential patterns that might influence the quality or nature of explanations. The clustering analysis revealed three well-defined clusters of methods: the first cluster *C-high* comprises 4,392 instances characterized by high metric values; the second *C-low* cluster includes 66,051 instances with low metric values; and the third *C-mid* cluster contains 29,468 instances with medium-level metrics.

We identified more instances in the *C-low* cluster, this outcome is particularly interesting, as it suggests that developers apply refactorings for a variety of reasons, not solely based on high values in code metrics. Such a domain-specific characteristic further underscores the importance of explainability. Developers often expect refactoring recommendations to target code snippets with high complexity or coupling metrics (TSANTALIS et al., 2020). When a recommendation is made for, say, a short and simple method, developers may naturally question its validity. In these cases, clear and trustworthy explanations become even more critical to support decision-making.

5.5.1.2 Instances to be Explained

For the analysis, we selected only instances from the C-low and C-hight clusters. Additionally, we filtered instances based on the model's prediction confidence, retaining

only those with a confidence level between 85–95% and those with 95% or higher. The confidence level represents the model's estimated probability for a given prediction. For example, if the model predicts class 1 (i.e., refactor) with a confidence of 90%, this can be interpreted as a strong indication that the instance should indeed be refactored.

We chose to include only RF predictions with a confidence level greater than 85%. This decision is again motivated by a domain-specific consideration. In our view, an IDE equipped with a refactoring recommender should generate suggestions only when the model exhibits a reasonably high degree of confidence. Otherwise, it could overwhelm the developer with excessive and unreliable recommendations, which may lead to mistrust in the tool.

Classification of the Instances (RF Precision x High/Low Values)		
	C-low values	**C-high values**
G1: RF Prec: 85-95%	1000 instances	1000 instances
G2: RF Prec: >95%	1000 instances	1000 instances

Table 13 – Number of Instances in Each Category

After applying these filters, we obtained a final dataset of $4,000$ instances, for which we generated explanations to support our empirical analysis. These instances were divided into four equally sized categories (1,000 instances each), as shown in Table 13. The categorization was based on two factors:

Model confidence level:

G1: Predictions of the Random Forest with confidence between 85–95%.

G2: Predictions of the Random Forest with confidence greater than 95%.

Cluster characteristics:

C-high: Instances from the cluster with high metric values (across the 23 metrics).

C-low: Instances from the cluster with low metric values (across the 23 metrics).

This stratification allowed us to systematically explore how both the model's confidence level and the structural characteristics of the code influence the generated

explanations. Considering these two dimensions, we were able to analyze whether explanations remain coherent across different levels of model certainty and code complexity. This approach also enabled us to assess whether certain types of methods (e.g., highly complex versus simpler ones) are more prone to divergent or less informative explanations, which is particularly relevant in the context of developer trust and the practical adoption of explainable refactoring tools.

5.5.2 Setting up the Prototype

The prototype was developed in Python, and it can be found in the GitHub repository (ADVANSE-LAB, 2024). The repository contains all the necessary instructions about what is mandatory and what must be configured. Besides, it provides examples of what can be sent to configure the Consensus Module with the different possible parameters. For the time being, our prototype does not have a graphical interface, nor is it implemented in the form of a plug-in ready to be attached to an IDE. As a result, it is necessary to pass the parameters as described in this article. To turn this prototype into a usable, ready-to-use tool (plug-in) we still need to develop a module that monitors and extracts metrics from the source code and passes these values on to this prototype. For the analysis, we customized our prototype as follows:

- **Internal Explainers**. We considered the 3 well-known state-of-the-art XAI Explainers: SHAP, LIME and Anchor;
- **Weight of the explainers**. We assigned the highest priority to Anchor, given its ability to generate more precise and faithful explanations. Anchor produces "anchors"—conditions that, when satisfied, guarantee the same prediction as the original model, offering a higher degree of reliability (RIBEIRO et al., 2018);
- **Strictness Level**. We set the *medium* level, which involves generating two sets of features: one containing attributes shared by all three explainers (full consensus), and another with attributes shared by at least two explainers (n-1 Explainers);
- **Number of Features in the Consensual Explanation**. We configured the prototype to generate explanations with $K = 1$ and $K = 3$ features. This choice aims

to identify the most influential variables in the model's decision process, producing concise, focused, and informative explanations.

Subsequently, we generated explanations for the 4000 instances and evaluated the (dis) agreement between the explanation methods using the *Feature Agreement* and *Rank Agreement* metrics.

5.5.3 Result

Figure 8 shows 16 matrices that illustrate the (dis)agreement between pairs of explanations. These matrices were generated by combining the parameters C-low and C-high, groups of RF precision (G1 and G2), the number of features (k) and the metrics Feature Agreement (RA) and Rank Agreement(RA). We will refer to each matrix by the number that identifies it.

The first 8 matrices, at the top (matrices from 1 to 8), were generated using the FA metric, while the bottom eight (matrices from 9 to 16) utilize the RA metric. In the matrices, known as heatmaps, lighter colors indicate strong disagreement, while darker colors indicate strong agreement. Because of space limitations, we concentrate our analysis on the most typical scenarios. Each subsection below focuses on a specific scenario.

5.5.3.1 Analyzing the FA metric.

For Heatmap 1, our approach achieves an average agreement of 0.78 (SHAP: 0.71, LIME: 0.84, and ANCHOR: 0.81) while EBM shows an average agreement of 0.70 (SHAP: 0.68, LIME: 0.69, and ANCHOR: 0.73). In heatmap 2, both our approach and EBM yield the same average agreement values as in heatmap 1. In heatmap 3, our approach shows an average agreement of 0.69 (SHAP: 0.72, LIME: 0.61, and ANCHOR:0.74) while EBM shows an average agreement of 0.56 (SHAP: 0.66, LIME: 0.54, and ANCHOR: 0.47). Similarly, in heatmap 4, both approaches replicate the results observed in heatmap 3.

For Heatmap 5, our approach achieves an average agreement of 0.49 (SHAP: 0.32, LIME: 0.49, and ANCHOR: 0.65) while EBM shows an average agreement of

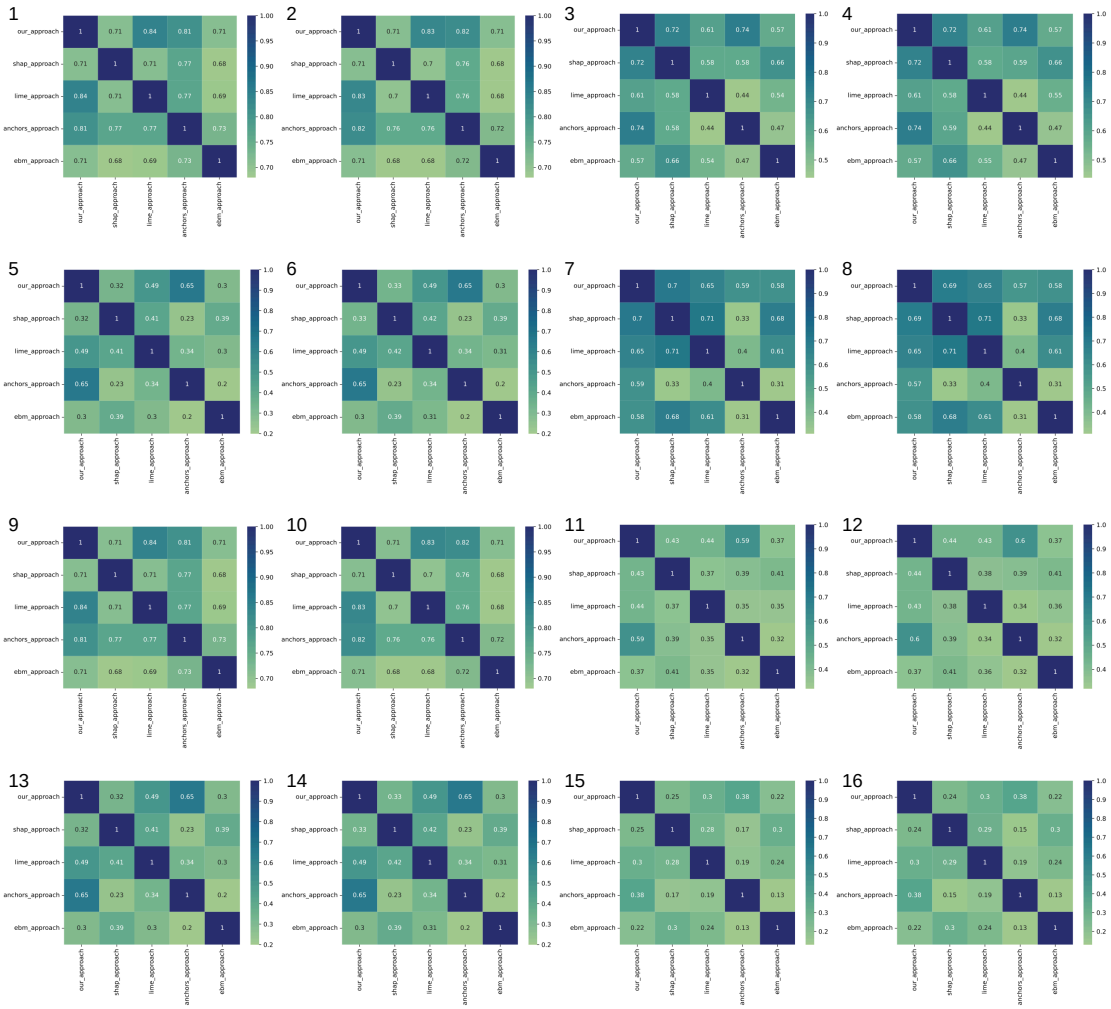


Figure 8 – Comparison among Explanation Approaches using HeatMaps

0.30 (SHAP: 0.39, LIME: 0.30, and ANCHOR: 0.20). In heatmap 6, both our approach and EBM yield the same average agreement values as in heatmap 5. In heatmap 7, our approach shows an average agreement of 0.64 (SHAP: 0.70, LIME: 0.65, and ANCHOR: 0.59) while EBM shows an average agreement of 0.53 (SHAP: 0.68, LIME: 0.61, and ANCHOR: 0.31). Similarly, in heatmap 8, both approaches replicate the results observed in heatmap 7. Table 14 shows the summary of this analysis.

Table 14 – Level of agreement and percentage improvement

Heatmap	Cluster	Our	EBM	improvement (%)
1	C-Low	0.78	0.70	11.4
3	C-Low	0.69	0.56	23.2
5	C-High	0.49	0.30	63.3
7	C-High	0.64	0.53	20.7

This analysis highlights that our approach consistently achieved higher agreement values (FA metric) compared to the EBM technique across the analyzed heatmaps. The percentage improvement ranged from 11.4% to 63.3%, with the most notable difference observed in the C-High cluster (heatmap 5). These results indicate that our method produces more consensual and stable explanations, especially in contexts with greater values of the features.

The analysis also reveals that there is no significant difference between categories G1 (Precision between 85-95%) and G2 (Precision greater than > 95%) when k is constant. This suggests that, despite variations in the precision of the Random Forest model, the explanation remains consistent. This is evident in heatmap 1 for G1 and heatmap 2 for G2, where our approach shows the same agreement with SHAP with 0.71, LIME with 0.84, and ANCHOR at 0.81. This trend continues in heatmaps 3 and 4, which present equal or very similar values. We believe that a precision above 85% already indicates high reliability, resulting in minimal or no variation in the features and their rankings used to generate the explanations.

5.5.3.2 Analyzing the metric FA when increasing the K-features.

We analyze the effect of increasing the number of features K=1 for k=3 on the agreement, considering the FA metric and both clusters (C-Low and C-Hight). As

previously shown, we do not differentiate between precision groups in this analysis, as the results are approximately equivalent. Therefore, there is no significant impact of model precision on the agreement outcome.

For scenario 1, we analyze the C-Low, comparing heatmaps 1 and 3. For scenario 2, C-Hight, we compare heatmaps 5 and 7. Thus, for scenario 1, our approach shows a minimum average reduction in agreement of 0.09 which represents 12.28% while the EBM shows an average agreement reduction of 0.14, representing 20.48%. For scenario 2, our approach shows an average agreement increase of 0.16, which represents 32.88% while the EBM shows an agreement increase of 79.58%. Although the EBM method presents a higher percentage of relative growth, the absolute values achieved by our approach are consistently higher (for K=3, 0.65 compared to 0.53 for EBM). This indicates that, despite a smaller relative variation, our method starts from a more solid base and achieves higher performance in absolute terms. The analysis reveals that our proposal maintains, on average, good agreement with SHAPE, LIME and ANCHOR, outperforming the EBM technique.

5.5.3.3 Analyzing the metric FA when varying from C-Low to C-High.

Considering k =1, we analyze whether there is a difference in agreement when varying between the C-Low and C-High clusters. To do that, we examine the heatmaps 1 and 5.

For the C-Low cluster, heatmap 1, our approach demonstrates a high average agreement of 0.79 (SHAP: 0.71, LIME: 0.84, and ANCHOR: 0.81) whereas EBM shows an average agreement of 0.7 (SHAP: 0.68, LIME: 0.69, and ANCHOR: 0.73). In contrast, for the C-High cluster, heatmap 5, our approach yields an average agreement of 0.49 (SHAP: 0.32, LIME: 0.49, and ANCHOR: 0.65), while EBM achieves an average agreement of 0.30 (SHAP: 0.39, LIME: 0.30 and ANCHOR: 0.20). This analysis reveals that our approach maintains a higher average agreement than EBM when the clusters vary from C-Low to C-High. Noticeably, in the C-High cluster, the agreement values are lower. We attribute this decline to the increasing complexity of the methods within the C-High cluster. This complexity leads to a greater number of features available for generating explanations, which affects the significance of each individual feature's contribution.

5.5.3.4 Analyzing the agreement level considering FR metric.

We analyze the agreement focusing on Feature ranking (FR) metric. In the K=1 scenario, regardless of the group (G1 and G2) and the cluster (C-Low and C-High), we observe that the FR and FA metrics yield identical values. This means that heatmaps 9 and 10 are equal to heatmaps 1 and 2, and heatmaps 13 and 14 are equal to heatmaps 5 and 6. This occurs because only one feature is analyzed, resulting in coinciding agreement and consequently identical ranking. Therefore, the analysis previously conducted for the FA metric also applies to FR in this case.

When analyzing the heatmaps for the FR metric at K=3 and considering both clusters, we do not take the groups into consideration, as they have no impact on the explanation. We observe that the quadrant formed by heatmaps 11, 12, 15, and 16 exhibits the lowest agreement compared to the other quadrants. This indicates a higher level of disagreement among the explanations. Furthermore, heatmaps 15 and 16, which correspond to the C-High clusters, show even lower agreement values. These results suggest that the FR metric is sensitive not only to the increase in K but also to the complexity associated with the Cluster type. Nevertheless, our proposal consistently achieves better agreement values than the EBM technique. This is particularly evident in heatmap 15, where our method achieves an average agreement of 0.31 (SHAP: 0.25, LIME: 0.3, ANCHOR: 0.38) while EBM reaches only 0.22.

5.6 Final Considerations

This chapter introduced a strategy to address the problem of "disagreement"among explainers, resulting in the Consensual Explainer module. The approach integrates ANCHOR, LIME, and SHAP, with configurable parameters such as the choice and number of explainers, the number of features in the final explanation, the strictness level, and explainer weighting. An empirical evaluation using EBM as a baseline showed that our method consistently achieved higher agreement values, particularly for Feature Agreement (FA) and Rank Agreement (RA), with improvements ranging from 11.4% to 63.4% across clusters. These findings indicate that the proposed approach generates more stable and reliable explanations, especially in scenarios with higher feature complexity.

Chapter 6

AN APPROACH FOR GENERATING COMPLETE RECOMMENDATION OF EXTRACT METHOD

6.1 Initial Considerations

Vast tools in the literature have explored the use of metrics to recommend refactorings. However, few studies have leveraged the contextual semantic relationships between source codes to improve refactoring recommendations. This chapter presents the methodological process conceived for the development of the refactoring recommendation approach, detailing the design rationale, implementation challenges, and the integration of its main components. The methodology involves generating candidate snippets, computing the affinity between code fragments with a refined CodeBERT model, producing explanations through the Consensual Explainer, and leveraging the GPT model to integrate and refine these results. The outcome is a set of recommendations that are complete and understandable for developers and more likely to be accepted.

6.2 Detailing the Methodological Process

The refactoring recommendation approach for Extract Method aims to generate *complete* recommendations grounded in *real extractions* while leveraging the semantic representation of the code. This is achieved through a pipeline that integrates multiple specialized components, whose interaction results in a cohesive and unified approach.

The term *real extraction* denotes that the recommendations are grounded in the

analysis of real software projects, capturing the genuine motivations behind developers' refactoring decisions. In the literature, recommendation approaches usually aim to solve specific problems, such as code smells; however, the motivations for applying a refactoring can be manifold. For instance, Henrique et al. (HENRIQUE et al., 2021) mined commits to discover the real reasons why developers apply the extract method, and they found 11 reasons. Thus, 6 of these 11 motivations are related to the intention of improving the quality of the existing code. For example, removing duplicate code, improving readability, and reducing the size of the method. The remaining 5 intentions are related to bug removal, adding features, and improving performance.

The term *complete* recommendation means that the recommendation must supply all the elements to support the decision-making process of the user. Regarding this point, in Chapter 4, we specify the following four elements that all recommendations must have: (i) Which refactoring should be applied; (II) Where the refactoring must be applied; (III) Why the refactoring was recommended; (IV) Benefits resulting from applying the refactoring. The *why* element is essential for making recommendations understandable, as it clarifies the reasons why the model recommends refactoring. Providing explanations in natural language can increase developers' trust and engagement in the recommendation. By incorporating all elements, the tool aims to deliver a complete and relevant single recommendation rather than a long list, as is common in most existing approaches.

Since our goal is to formulate a *Complete Recommendation*, each phase of the process is designed to fulfill one or more of the criteria of a recommendation. Figure 9 shows an overview of the process followed to develop the approach, which is structured as a multi-phase pipeline divided into four phases: (I) Building the Extract Method Dataset; (II) Generating the Recommendation Model; (III) Building the Consensual Explainer; and finally (IV) Assembling the Recommendation Pipeline. These phases are tightly connected to the three central criteria of a recommendation: *Where*, *Why*, and *Benefits*. Phases I and II ensure the identification of the fragment to be extracted (*Where*); Phase III provides the rationale of the model behind the recommendation (*Why*); and Phase IV integrates all components to deliver the complete recommendation, including its *Benefits* and a sketch of its application.

Phase I and Phase II are directly linked to the *Where* criterion. In Phase I, we construct the Extract Method dataset called *EM Dataset* by collecting positive and

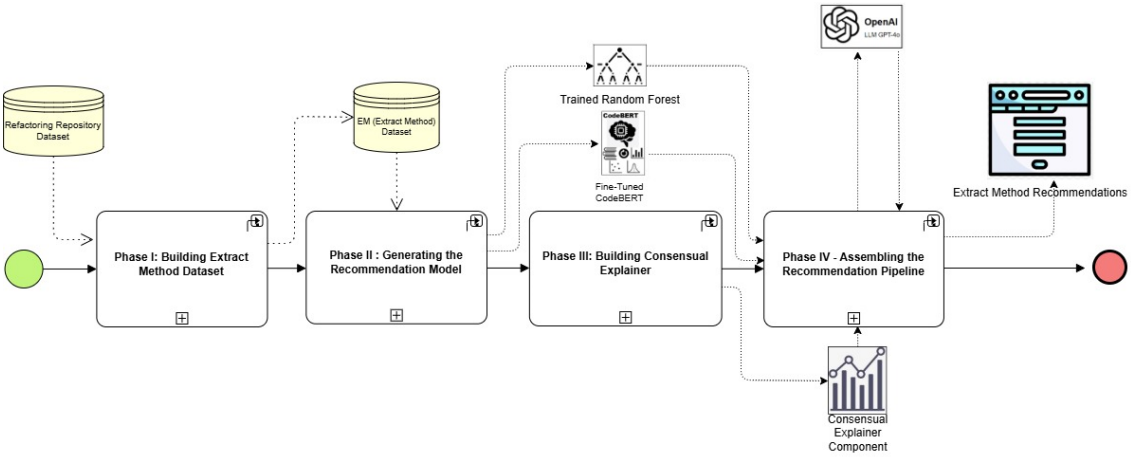
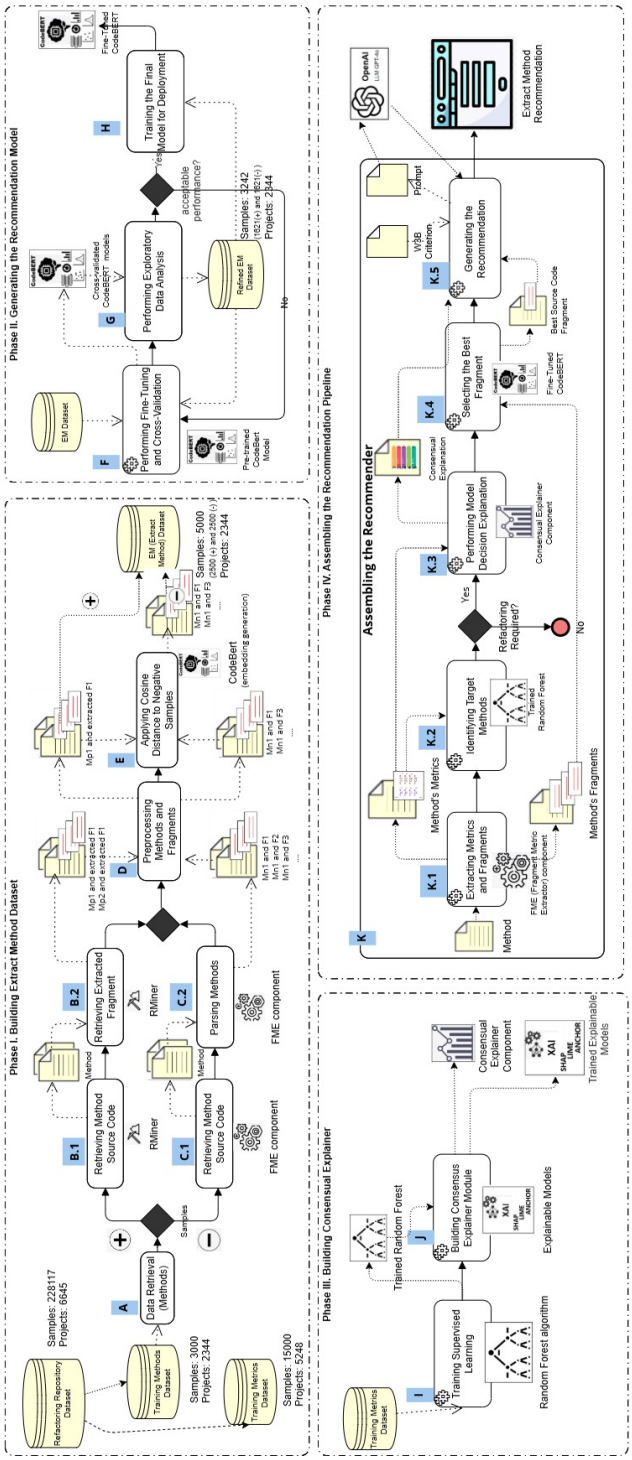


Figure 9 – Overview of the Refactoring Recommendation Tool

negative samples of Extract Method refactoring from *Refactoring Repository Dataset* (ANICHE et al., 2019). Positive samples are methods that underwent Extract Method in the past. They were obtained by retrieving the source code of the methods and their corresponding extracted fragments. For negative samples, we retrieved methods that had not undergone Extract Method in the past. They were retrieved from the same initial repository, but only the most representative ones were selected using a cosine similarity strategy.

Phase II focused on fine-tuning the Pre-trained CodeBERT model using the created *EM Dataset* to distinguish between fragments that should or should not be extracted. By evaluating the model through cross-validation and performing exploratory analysis of feature distributions, this phase consolidates the ability to identify the fragment to extract, which is the core of the *Where* criterion.


Phase I. Building Extract Method Dataset
Phase II. Generating the Recommendation Model
Phase III. Building Consensual Explainer
Phase IV. Assembling the Recommender Pipeline
Assembling the Recommender
Refactoring Repository Dataset
Samples: 228417
Projects: 6645
Data Retrieval (Methods)
Training Methods Dataset
Samples: 3000
Projects: 2344
Training Metrics Dataset
Samples: 15000
Projects: 5248
Retrieving Method Source Code
Retrieving Extracted Fragment
Parsing Methods
Preprocessing Methods and Fragments
Applying Cosine Distance to Negative Samples
CodeBert (embedding generation)
EM Dataset
Performing Fine-Tuning and Cross-Validation
Performing Exploratory Data Analysis
Training the Final Model for Deployment
Refined EM Dataset
Samples: 3242
Projects: 2344
Training Supervised Learning
Random Forest algorithm
Trained Random Forest
Building Consensus Explainer Module
Explainable Models
Consensual Explainer Component
Trained Explainable Models
Extracting Metrics and Fragments
Identifying Target Methods
Performing Model Decision Explanation
Selecting the Best Fragment
Generating the Recommendation
Refactoring Required?
Extract Method Recommendation

Phase III is responsible for addressing the *Why* criterion. This phase introduced a consensus explainer module built upon multiple explainable AI models such as SHAP, LIME, and ANCHOR, alongside the random forest model to provide interpretable decisions. This phase provides interpretable rationales behind each recommendation, improving understanding and creating a bond of trust between the user and the recommendations.

Phase IV brings the process to completion by assembling the recommendation pipeline and connecting it to the *Benefits* criterion. The developed components: the Fragment Metrics Extractor (FME), the trained Random Forest, the fine-tuned CodeBERT model, and the consensual Explainer are integrated into a unified approach. Additionally, the GPT model is incorporated to refine the complete Extract Method Recommendation along with explanations in natural language. This integration not only delivers the complete refactoring but also provides a sketch of its practical application, thereby completing the link between the criteria and the recommendation.

Figure 6.2 presents the complete process, the four phases, and their respective steps. The following subsections describe each of them in detail.

6.2.1 Phase I: Building Extract Method Dataset

This phase is mainly responsible for the correct formulation of the *Where* criterion. Achieving this requires a dataset composed of code pairs, consisting of methods and their fragments. We argue that the source code of a method has a rich semantic relationship with all the fragments that compose it, and this relationship can be exploited to identify the best candidate for extraction. A similar approach was proposed by Wenhao (MA et al., 2023), who constructed a dataset pairing methods and classes to detect Feature Envy.

However, existing datasets are insufficient for our purposes, as they do not provide paired representations of methods and their corresponding extracted fragments; instead, they rely mainly on metrics. To overcome this limitation, we constructed a specialized dataset, which we call *Extract Method Dataset*. This dataset is used in Phase II to train the CodeBert model, enabling it to explore the relationship between the source code of the method and their fragments. In our context, the fragment with the weakest relationship to the rest of the method can indicate the most suitable candidate for extraction. This identification contributes to formulating a complete recommendation for the Extract

Method, since the *where* component is primarily determined by the fragment selected for extraction.

The *Extract Method Dataset* is structured as follows:

$$\begin{aligned} \text{Sample} &= \langle \text{SCMethod}, \text{SCFragment}, \text{Label} \rangle; \\ &\text{where} \\ \text{SCMethod} &:= \text{Source Code of the Method}; \\ \text{SCFragment} &= \text{Source Code of the fragment}; \quad \text{and} \\ \text{Label} &= \{0 - 1\} \end{aligned} \tag{6.1}$$

To ensure the quality of the dataset, the construction process involved the careful selection and treatment of both positive (Steps B.1 and B.2) and negative (Steps C.1 and C.2) samples. This phase is crucial to ensure that the recommendation approach can learn from real development practices and distinguish meaningful fragments from arbitrary or less relevant ones. As a result of this Phase I, we obtained a specialized and balanced *EM Dataset* with 5000 instances, 2500 positives, and 2500 negatives, covering 2344 different projects.

Step A. Data Retrieval (Methods)

The *Refactoring Repository Dataset* used in this research was originally compiled by Aniche et al. (ANICHE et al., 2020). It includes process, class, and method level metrics associated with different types of refactorings. For the purpose of this work, we focus exclusively on the Extract Method refactorings. In this dataset, the positive samples are methods that underwent extract method refactoring; identified using the RefactoringMiner (Rminer) tool (TSANTALIS et al., 2020), which detects refactoring operations across the history of the repository. On the other hand, for negative samples, the authors adopted the strategy of selecting methods that did not undergo refactoring operations for 50 consecutive commits.

We select 18000 samples from the *Refactoring Repository Dataset* to construct two specialized datasets: one called *'Training Methods Dataset'* for training the Pre-

trained CodeBERT model in Phase II, and another called *'Training Metrics Dataset'* for training the Random Forest model in Phase III.

To construct the *'Training Methods Dataset'*, we applied the following criteria: only methods that underwent the Extract method refactoring type, belonged to projects publicly available on GitHub, and were developed in Java. Additionally, we prioritized maximizing the number of distinct projects to ensure a diverse range of development contexts and reduce bias toward specific projects. As a result, the final dataset comprises 3000 samples, 2000 positive and 1000 negative instances.

Regarding the features included in the dataset, the following were selected: *Parentcommit* (the commit of the method before the extraction); *ChildCommit* (the commit of the method after the extraction); *Commit* (the commit of the non-refactored method), *gitUrl* project's URL; *filePath* (the location of the method within the project); and *shortMethodName*. These features allow us to locate the method identified as positive or negative, ensuring that the correct method and fragment (in the case of the positive ones) are returned for analysis.

To make the distinction between strategies employed for positive and negative samples, the process is divided into two ways: Steps B.1 and B.2 for positive samples, and Steps C.1 and C.2 for negative ones. Each step is described below.

Step B1. Retrieving Method Source Code

Main Objective. The objective of this step is to locate the source code of the method considered a positive sample and retrieve it, storing the source code in a JSON file for use in subsequent stages of processing and analysis.

Technical Process. The *Training Methods Dataset* is processed to extract the complete source code of each method (positive samples). The focus is on the version of the method before the Extract Method refactoring. To achieve this, the process uses the *gitUrl* project's gitURL and the *ParentCommit* (THE commit that precedes the refactoring), along with the RefactoringMiner tool, which precisely identifies the method's location within the source file. Once located, the source code of the method is retrieved directly from GitHub through the use of the raw content endpoint, allowing access to the exact version of the file at that commit. This automated retrieval ensures consistency and reproducibility across all analyzed examples.

Step B2. Retrieving Extracted Fragment

Main objective. This step aims to locate the source code fragment extracted from the Method in *Step B1* and retrieve it, storing the fragment in the same JSON file. As a result, each row in the file contains both the original method and the extracted code fragment.

Technical Process. This step builds upon step B1 by retrieving the code fragment that was extracted from the original method and transformed into a new method as a result of the Extract Method refactoring. To perform this task, the process accesses *ChildCommit* (the commit after the refactoring was applied) along with the RefactoringMiner tool; the refactored code fragment is accurately located and extracted.

Step C1. Retrieving Method Source Code (Negative Samples)

Main objective. The objective of this step is to locate the source code of the method considered a negative sample and retrieve it, storing the source code in a JSON file for use in subsequent stages of processing and analysis.

Technical Process. The *Training Methods Dataset* is processed to extract the complete source code of each method for the negative samples. The method's source code is retrieved directly from the GitHub repository. The features used for this process are: *Commit*, *gitUrl*, *filePath*, and *shortMethodName*. The process starts by constructing a raw content URL that points to the exact version of the file in which the method is located, using GitHub's raw interface.

An HTTP request is issued to download the source file corresponding to the specified commit. Once the file is retrieved, the JavaParser library performs syntactic analysis and generates an Abstract Syntax Tree (AST) of the code. This AST is then traversed to locate the method that matches the given *shortMethodName*. This step ensures that the method is extracted exactly as it existed in that historical version of the repository.

Step C2. Parsing Methods

Main objective. The objective of this step is to parse the method and identify relevant source code fragments. Since each method typically contains multiple structures (for, if , while, etc.), several fragments are generated per method. Consequently,

this process produces multiple samples, one for each valid method-fragment pair. The step produces a JSON file containing each complete method along with its associated fragments.

Technical Process. This step builds upon step C1. After retrieving the source code file, the method of interest is parsed using the JavaParser library, which performs syntactic analysis of the Java source code and constructs an Abstract Syntax Tree (AST) that facilitates the identification of the target method based on the *shortMethodName*. Once the method node is located within the AST, the parser traverses its body to extract individual statement-level fragments. To ensure semantic relevance, statements that are empty, comments, or represent only structural delimiters (e.g., opening or closing braces) are excluded.

Step D. Preprocessing Method and Fragments

Main objective. To ensure the quality and representativeness of the dataset used for training the CodeBERT model in *Phase II*, several preprocessing steps were applied. These actions aimed to remove noise, eliminate redundancy, and retain only relevant code units for the recommendation task.

Technical Process. The preprocessing task included the following actions:

- *Filtering Class and projects*. We selected only one method per distinct class and project. This decision helps reduce bias introduced by multiple similar methods from the same source, which could otherwise skew the model's learning.
- *Check if the fragment is equal to the entire method*. We checked whether any extracted fragment was identical to its corresponding full method. This situation can occur when the method is particularly short or when multiple extractions overlap entirely with the original method. In such cases, we excluded these fragments, as they do not constitute meaningful partial extractions. This filter was applied only to the positive samples, since full-method fragments can be considered valid negative examples.
- *Fragment Coverage Filtering*. Fragments whose length was greater than or equal to 80% of the full method body were excluded, as they often did not represent meaningful partial extractions.

- *Duplicate Removal*. To avoid data redundancy, we identified and removed duplicate entries in the dataset. This process included both full methods and extracted fragments that appeared multiple times.

- *Code Cleaning*. All code samples underwent standardization, including the removal of single-line (//) and multi-line (/* ... */) comments, elimination of annotations such as @Override, @Test, and others, as well as the removal of excessive whitespace, line breaks, tab characters, and multiple consecutive spaces.

- *Fragment-Method Separation*. After identifying the relevant code fragments, each fragment was programmatically separated from its original method body. This step was critical to prevent data leakage, as retaining the full method alongside its fragment could inadvertently allow the model to learn from redundant contextual information. By isolating the fragments, we ensured that the input fed into the model represents realistic, independent scenarios of partial code, thereby enhancing the generalization capability of the recommender.

Step E. Applying cosine distance strategy for negative samples

Main objective. The objective of this step is to identify the most representative negative samples. These are the samples that exhibit the greatest semantic divergence from the positive samples. In other words, they correspond to the examples that share the least meaning with positive instances, so that the negative set contains clear counterexamples rather than borderline cases, ensuring a more representative dataset.

Technical Process. Cosine distance (SALTON et al., 1975) is employed in this step to identify the *most representative negative samples*. This metric quantifies semantic divergence by measuring the angle between two vectors in the embedding space, so a higher cosine distance indicates a greater disparity between samples. To obtain the vector representations, we leverage the pre-trained CodeBERT model to encode source code elements into dense embeddings that capture both syntactic and semantic meanings.

To implement this strategy, both the original source code method *SourceCodeMethod* and its corresponding extracted fragment *fragment* are tokenized and passed through CodeBERT. The resulting embeddings are then combined by summing the method and fragment vectors, forming a joint representation that captures both the con-

text of the original method and the semantics of the extracted fragment. This process is applied consistently across all entries in both the positive and negative sets.

The computation of the metric was performed using the implementation available in scikit-learn (PEDREGOSA et al., 2011). The Cosine distance between two vectors $\vec{a}$ and $\vec{b}$ is defined as:

$$\boxed{\text{CosineDistance}(\vec{a}, \vec{b}) = 1 - \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \cdot \|\vec{b}\|}}$$

For each negative sample, the minimum distance to all positive samples is computed:

$$\boxed{d_{min}(\vec{n}) = \min_{\vec{p} \in P} \text{CosineDistance}(\vec{n}, \vec{p})}$$

where $\vec{n}$ is the embedding of a negative pair and P is the set of embeddings of all positive pairs.

We select the N most distant negative examples, i.e., those with the highest $d_{min}(\vec{n})$. In the current configuration, we define $N = 2500$, thus selecting the top-2500 negative samples with the highest semantic divergence from the positive set.

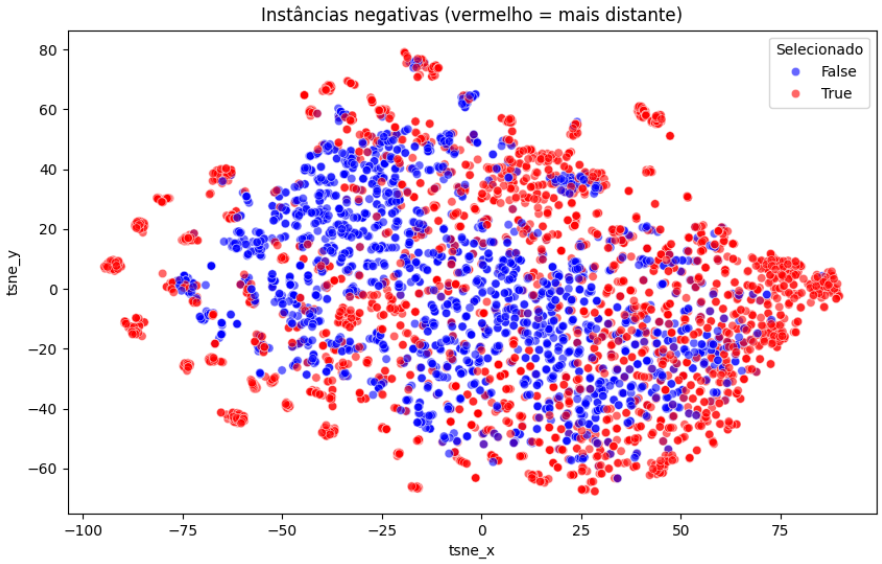


Figure 10 – 2D visualization of negative instances using t-SNE

Figure 10 shows a two-dimensional visualization of the negative instances using t-distributed Stochastic Neighbor Embedding (t-SNE), which reduces the high-dimensional embeddings into a 2D space for easier interpretation. Each point represents a negative example, where the blue points correspond to all available instances, and the red points highlight those that were selected for being the most distant from the positive class. The plot helps illustrate how the selected negatives are spread across the space, typically appearing more isolated or on the edges, which suggests that they may carry more distinctive information for model training.

As an outcome of this phase, we constructed the *Extract Method Dataset*, comprising 5000 instances equally distributed between positive (2500) and negative (2500) samples. By leveraging semantic-based selection rather than random sampling, this process enhances both the robustness and the generalization capability of the recommendation model to be trained in Phase II. The notebooks utilized in this study are publicly available in the replication repository (ANGULO, 2025a).

6.2.2 Phase II: Generating the Recommendation Model

This phase, together with Phase I, focuses on accurately identifying the *Where* criterion. While Phase I involved constructing a specialized *EM dataset* of code pairs, each consisting of a method and its fragments. Phase II is dedicated to training the CodeBERT model on this dataset to determine the most suitable code snippet for extraction.

The pre-trained CodeBERT model was selected as the base model for this approach due to its ability to capture deep semantic relationships between source code (FENG et al., 2020). Unlike traditional metrics, which rely on structural code features, CodeBERT learns contextual representations of code snippets, enabling the assessment of semantic relationships between methods and fragments; precisely the capability explored in this work.

Step F. Performing Fine-Tuning and Cross-Validation

The fine-tuning step aims to adapt a pre-trained CodeBERT model to the specific task of identifying the most relevant fragments for extraction. This process specializes the model's general understanding of programming languages to the context of Extract Method refactoring in Java. To achieve this, the model is trained on the *Extract Method*

Dataset constructed in Phase I, which contains labeled examples representing both positive and negative cases.

We formulate the Extract Method recommendation task as a regression problem. Thus, the model is able to assign a continuous affinity score that captures relationships between the original method and the candidate fragment. This approach allows for the ranking of multiple extraction candidates, providing developers with a more informative and flexible recommendation system rather than a rigid binary decision. The result of this step is a fine-tuned version of CodeBERT, specifically adapted to evaluate fragment candidates in real-world code contexts.

The training setup was carefully configured to reflect the task's regression nature. Each model was trained for five epochs, with a small batch size of two per device to accommodate memory constraints and encourage more frequent updates. We used Mean Squared Error (MSE) as the primary evaluation metric and checkpointed the model based on the lowest validation MSE. All the parameters are shown in 6.1.

To evaluate the performance and generalization capacity of the fine-tuned CodeBERT model, a 5-fold cross-validation strategy was employed. The dataset was split into five distinct partitions, with each fold serving once as the validation set while the remaining four folds were used for training. For each iteration, a fresh instance of the CodeBERT model was initialized and trained, preventing contamination of learned parameters across folds. This procedure ensured a robust and unbiased assessment of the model's ability to generalize across different code samples and refactoring contexts.

```
training_args = TrainingArguments(
    output_dir=fold_output_dir,
    logging_strategy="epoch",
    eval_strategy="epoch",
    save_strategy="epoch",
    per_device_train_batch_size=2,
    per_device_eval_batch_size=2,
    num_train_epochs=5,
    logging_dir=os.path.join(fold_output_dir, "logs"),
    report_to="none",
    save_total_limit=2,
    load_best_model_at_end=True,
    metric_for_best_model="mse",
    greater_is_better=False
)
```

Listing 6.1 – Training configuration using HuggingFace's TrainingArguments

Additional regression metrics such as Mean Absolute Error (MAE) and R^2 (coefficient of determination) were also computed to provide a more comprehensive understanding of the model's prediction quality. Although the task was modeled as a regression problem, we extended the evaluation to include classification-oriented metrics by applying a 0.65 threshold to the predicted affinity scores (this value of 0.65 was identified after multiple iterations of the steps in this Phase). This allowed us to calculate precision, recall, F1-score, and Area Under the ROC Curve (AUC), offering a dual perspective on the model's performance.

The final results across all folds were saved individually and can be aggregated to compute macro-level statistics. This hybrid evaluation strategy(combining regression and classification metrics) provides a more nuanced and flexible understanding of the model's effectiveness in identifying suitable code fragments for Extract Method refactoring.

This phase is iterative and was run until acceptable model performance was achieved. For the last iteration, we present the quantitative summary of Cross-Validation results; the performance across the five folds demonstrates that the fine-tuned model achieves stable and reliable predictions as shown in Table 15. The *Mean Squared Error (MSE)* ranged from 0.152 to 0.163, indicating low average squared differences between predicted and true affinity scores. The R^2 values, between 0.346 and 0.386, suggest a moderate proportion of variance explained by the model, which is a strong result given the complexity and noise inherent in real-world refactoring data. The *Mean Absolute Error (MAE)* remained consistently around 0.37–0.38, confirming that the model's average deviation from the ground truth was small.

From a classification perspective, derived by thresholding the continuous outputs, the model achieved exceptionally high *precision* (min: 0.978) and *recall* (min: 0.996) across all folds. The *F1-scores* were all above 0.98, and the *AUC*(Area Under the ROC Curve) exceeded 0.997 in every fold, reflecting separability between positive and negative samples. These metrics indicate that, even when viewed as a binary decision system, the model maintains excellent discriminatory power.

These results confirm that modeling the Extract Method recommendation as a

regression task not only provides fine-grained output but also achieves high classification performance when needed. The model generalizes well across code samples and preserves both precision and robustness in identifying relevant code fragments.

Table 15 – Summary of cross-validation metrics across five folds.

Metric	Mean	Min	Max
Mean Squared Error (MSE)	0.159	0.152	0.163
R^2 Score	0.363	0.346	0.386
Mean Absolute Error (MAE)	0.379	0.372	0.384
Precision	0.985	0.978	0.997
Recall	0.999	0.997	1.000
F1-Score	0.992	0.987	0.997
Area Under Curve (AUC)	0.999	0.998	1.000

Step G. Exploratory Data Analysis (EDA)

The Exploratory Data Analysis (EDA) step aimed to gain insights into the behavior of the refactoring recommendation model through visual and statistical investigations of the training and validation results.

Several visual tools and metrics were employed, including:

- Confusion Matrices: Used to visualize the model's performance in terms of correctly and incorrectly classified fragments selected for extraction.
- Learning Curves: Plotted to monitor the model's training progress across epochs, highlighting trends in both training and validation loss and accuracy.
- Histograms: Used to examine the distribution of predictions and probabilities associated with each class, as well as the frequency of samples across categories.
- ROC and AUC Curves: Provided a measure of the model's discriminative ability, visualizing the trade-off between true positive and false positive rates.

During training, the indices of the samples used in each fold were saved, enabling a deeper analysis of the specific instances involved in model learning. This allowed for a detailed understanding of which samples were contributing positively or negatively to the model's performance. Observations from this phase also provided important quantitative and qualitative insights, helping to identify data issues that led to prediction errors. This analysis contributed to building a more robust and reliable basis for evaluating and improving the refactoring recommendation system. The evaluation results are publicly available in the replication repository (ANGULO, 2025a).

Step H. Training the Final Model for Deployment

After the iterative process of fine-tuning the CodeBERT model and performing exploratory data analysis, the Refined *Extract Method* Dataset was constructed, consisting of 3242 samples from 2344 distinct projects. In this final step, the entire refined dataset is employed to train the definitive version of the model without partitioning for cross-validation. The purpose of this training is to consolidate the knowledge acquired in the previous iterations, maximizing the use of all available data to obtain a robust model suitable for deployment.

The Python project used for these steps, as well as the evaluation results, is publicly available in the replication repository (ANGULO, 2025a).

6.2.3 Phase III: Building Consensual Explainer

With the intention of formulating a complete refactoring recommendation, Phase III provides the model's reasoning behind the recommendation, which constitutes the *Why* criterion. To develop the explanation, we rely on explainability models that have gained significant attention in recent years.

In this phase, we addressed two major challenges. The first relates to the need to provide explanations for the decision to refactor a method, specifically clarifying the outputs generated by the CodeBERT model. Although CodeBERT demonstrates high predictive performance, it operates as a black-box model, making its internal decision-making process unintelligible to users. Introducing explainability at this stage enhances transparency and fosters trust in the recommendation results (GUNNING et al., 2019) (LONGO et al., 2020).

The second challenge concerns the issue of *disagreement among explainers* , which arises when different explanation techniques (e.g., SHAP, LIME, ANCHOR) yield conflicting justifications for the same prediction. These disagreements can manifest in three key ways: feature disagreement, where the top-k features identified as important by one explainer differ from those identified by the other; rank disagreement, where the ranking of the top-k features in terms of importance varies between the explainers; and sign disagreement, where the sign of the importance (positive or negative) assigned to the features is inconsistent across the explainers (ROY et al., 2022b) (PIRIE et al., 2023).

To address this limitation, we propose a consensus-based explanation strategy that aggregates the outputs of multiple explainers to generate a unified and coherent explanation for each recommendation. This strategy was operationalized through the implementation of a customizable tool, as further detailed in Chapter 5.

Step I. Training Supervised Learning

Given that transformer-based models such as CodeBERT operate as black-box architectures with limited interpretability, directly extracting meaningful explanations from their internal attention mechanisms or embeddings remains a challenging task. Explainable AI (XAI) addresses this limitation by providing interpretability to complex models. In this work, we adopt *Model-Agnostic Methods* through the use of a surrogate model, which allows us to approximate the behavior of the black-box CodeBERT model using a simpler and interpretable model (VILONE; LONGO2020EXPLAINABLE, 2020) (KIM; KIM, 2022) .

The surrogate model serves as a simplified representation of the original complex model, trained to mimic its behavior as closely as possible. While it may not fully capture the performance of the underlying black-box model, it offers a trade-off by providing transparency and interpretability. This facilitates a better understanding of the decision-making process and supports more informed analysis by practitioners (SALIH et al., 2025).

For this purpose, we selected the Random Forest algorithm as the surrogate model. This choice was motivated by its robustness, interpretability, and effectiveness in handling high-dimensional datasets, which have been widely reported in studies related to software engineering and refactoring recommendation systems (CUI et al., 2023; PALIT

et al., 2023; CUI et al., 2022; LEIJ et al., 2021; YUE et al., 2018) . Random Forest models are particularly suitable for explainability purposes, as they are naturally compatible with *post-hoc* and *model-agnostic* explanation techniques. In this work, the trained RF model was used in conjunction with three widely recognized XAI algorithms: SHAP (SHapley Additive exPlanations) (LUNDBERG; LEE, 2017a), LIME (Local Interpretable Model-Agnostic Explanations) (DIEBER; KIRRANE, 2020), and ANCHOR (RIBEIRO et al., 2018) to generate local explanations based on feature importance.

The *Training Metrics Dataset* was used to train the Random Forest model. This dataset comprises $15,000$ samples and 20 features, representing structural and behavioral metrics extracted from methods that underwent Extract Method Refactoring. The features are as following: Number of anonymous classes; Number of assignment statements; Coupling Between Object(CBO); Number of comparison operations; Number of lambda expressions; Lines of Code (LOC); Number of loop structures; Number of mathematical operations; Maximum depth of nested code blocks; Number of numeric literals; Number of parameters declared; Number of parenthesized expressions; Number of 'return'; Response For a Class (RFC); Number of string literals; Number of inner subclasses; Number of 'try-catch' blocks; Number of unique identifier tokens; Number of local variables declared; and Cyclomatic complexity.

As output of this step, we obtain the *Trained Random Forest model*. This model plays a dual role: first, it is used to train the explanation techniques (SHAP, LIME, and ANCHOR); second, it serves as an initial binary classifier in **Phase IV**, acting as a preliminary filter to determine whether a given method requires refactoring. If the classification is positive, the process continues with the activities in the recommendation pipeline.

Step J. Building the Consensus Explainer Module

Chapter 5 presents the complete process of building the consensus module, designed to be both adaptable and ready for practical use.Although the module can be instantiated with any set of explainers, for the refactoring recommendation task we adopted the default configuration, which consists of the three models SHAP, LIME, and ANCHOR. We set K=5, meaning that the consensus explanation incorporates the five most representative features identified by the module. It is also worth noting that, for

this recommendation task, the dataset employed in this work incorporates 20 of the 23 metrics originally defined in the prototype design.

6.2.4 Phase IV: Assembling the Recommendation Pipeline

This phase integrates all the components developed in the previous phases to build the recommender system for *Extract Method*, responsible for generating complete refactoring recommendations.

Phase IV consists of a single step, named *Assembling the Recommender*, which orchestrates and sequentially connects the elements produced in earlier phases. Specifically, the recommender relies on the enhanced version of the FME (Fragment Metric Extractor) component, the trained Random Forest model, the Consensual Explainer module, the fine-tuned CodeBERT model, and the GPT OpenAI service. GPT is employed together with three artifacts: *Categories*, the W3B criteria, and a tailored Prompt. These artifacts are used to refine the recommendation process and to generate the *Benefits* criterion, outlined in general terms by the *Categories* artifact.

All sub-steps encompassed by *Assembling the Recommender* are described in detail below.

K.1 Extracting Metrics and fragments.

The process begins by extracting structural and behavioral metrics, as well as fragments from the input method. These metrics and fragments are computed using the FME (Fragment Metric Extractor) component. It is important to highlight this component, as it plays a key role in the fragmentation of the method.

FME (Fragment Metric Extractor) . The FME component needs two input parameters: *URL* (the GitHub URL of the Class) and *MethodName* (Name of the method) to produce two important artifacts: JSON file with fragments of the Method and a CSV file with the structural and behavioral metrics.

For fragment extraction, the component initially relied solely on JavaParser; however, as the complexity of extracting relevant fragment candidates became evident, the component had to be enhanced and evolved into a more robust solution. After internal testing, it became clear that the tool needed not only to identify control structures but also

to detect cohesive blocks of code that could serve as candidates for method extraction refactoring. To address this, a structured strategy was adopted, combining syntactic analysis of the source code via the JavaParser library with the identification of cohesive blocks of code within the control structures.

The *FME module* receives two input parameters: *URL* (the GitHub URL of the Class) and *MethodName* (Name of the method). Then, we use JavaParser to traverse the target method's body (represented by a BlockStmt object) and identify well-defined control structures, such as *if*, *for*, *while*, *do-while*, *switch*, and *try-catch*. These structures were selected because they typically encompass semantically meaningful code blocks, often associated with specific behaviors and relatively self-contained within the method.

Once these structures were identified, the corresponding inside block of code was selected while preserving its original form. In other words, the extracted fragments consisted of consecutive lines of code, respecting the boundaries of the control structure and maintaining the internal cohesion of the segment. The extraction aimed not merely to isolate individual statements, but to retain the integrity of the fragment as a logically unified entity potentially suitable for extraction.

Then, two additional filtering rules were applied to eliminate less useful fragments: (1) slices whose number of lines was equal to or greater than that of the original method were discarded, and (2) fragments containing reserved keywords such as *break* or *continue* were excluded (when the entire structure is not part of the block), as these may indicate dependencies on external structures or hinder safe refactoring.

In addition to structural analysis, we measure a new variable called *totalUsageOutsideSlice*, which evaluates the potential coupling of the variables in each fragment. To calculate the variable, all the variables within the fragments are identified and their occurrences in the remainder of the method are counted. The value of the variable aims to estimate the degree of coupling between the fragment and the rest of the code, signaling whether the variables used in the slice remained relevant outside of it. Fragments with high reuse of external variables indicate potential high coupling, which may hinder their extraction as independent methods. The calculation is presented as follows:

$$\textbf{totalUsageOutsideSlice} = \sum_{v \in \text{slice}} \Big(\text{totalInMethod}(v) - \text{inFragmentCount}(v) \Big)$$

Where:

v	Each variable present in the fragment
totalInMethod(v) :	Total occurrences of the variable in the method
inFragmentCount(v):	Occurrences of the variable inside the fragment

The output is a JSON file containing the keys: *methodCode*,*sourceCodeFragment*, *totalUsageOutsideSlice*. The format is shown below:

```
[
 {
  "methodCode": "@Override public void afterTextChanged(Editable s){...}",
  "sourceCodeFragment": "boolean found = false;",
  "totalUsageOutsideSlice": 2
 },
 {
  "methodCode": "@Override public void afterTextChanged(Editable s) {...}",
  "sourceCodeFragment": "if (s.charAt(i) == '\n') {found = true; break;}",
  "totalUsageOutsideSlice": 3
 },
 {
  "methodCode": "@Override public void afterTextChanged(Editable s) {...}",
  "sourceCodeFragment": "mMultiline = false; int pos = mEditText.getSelectionStart()
      ;...",
  "totalUsageOutsideSlice": 0
 }
  ...
]
```

Listing 6.2 – Example of extracted fragments in JSON format

For the computation of structural and behavioral metrics, the *FME module* leverages the CK Metrics library (ck.jar) introduced by Maurício Aniche (ANICHE, 2015). This module performs an analysis on the same method that has been previously fragmented, deriving a total of 20 structural and behavioral metrics. The results are exported to a CSV file, structured in the following format:

```
1 id_,methodAnonymousClassesQty,methodAssignmentsQty,methodCbo,methodComparisonsQty,
      methodLambdasQty,methodLoc,methodLoopQty,methodMathOperationsQty,
      methodMaxNestedBlocks,methodNumbersQty,methodParametersQty,
      methodParenthesizedExpsQty,methodReturnQty,methodRfc,methodStringLiteralsQty,
      methodSubClassesQty,methodTryCatchQty,methodUniqueWordsQty,methodVariablesQty,
      methodWmc
2 1,0,5,1,1,0,17,1,1,3,2,1,1,0,6,0,0,0,14,3,5
```

Listing 6.3 – Example of CSV file metrics

In summary, the *FME module* is designed to ensure that the extracted slices correspond to functionally cohesive blocks, while simultaneously computing method-level structural and behavioral metrics. Each execution of the module produces two output files: one containing the method fragments in JSON format, and another containing the associated method metrics in CSV format. This design allows for the correct execution of subsequent stages of the pipeline.

6.2.4.1 K.2 Identifying Target Methods

In this sub-step, the Random Forest model trained in Phase II is responsible for identifying whether a method is a candidate for Extract Method refactoring. The model was trained on a dataset of software metrics derived from methods that had undergone this refactoring, thus capturing patterns in metric space that are indicative of suitable extraction opportunities.

The model acts as a binary classifier that evaluates the metrics of a method and decides if it should be targeted for refactoring. A probability threshold of 0.5 is applied: if the score is ≥ 0.5, the method is flagged as a refactoring target and subsequent steps are executed; otherwise, the pipeline terminates at this stage, avoiding unnecessary computations. Beyond classification, the Random Forest is also reused in Step K.3 together with explainability models to provide rationales for its predictions. This dual role ensures coherence in the approach, since the same model both identifies candidate methods and contributes to explaining the recommendation.

As input, this sub-step receives a CSV file containing the extracted metrics of the analyzed method. These features reflect structural and behavioral properties of the method. In this way, only methods meeting the criteria advance in the pipeline.

It is important to note that the identification of the target method in this step directly constitutes part of the *Where* criterion, which is essential for generating a complete recommendation.

K.3 Performing Model Decision Explanation

In this sub-step, the focus shifts to generating explanations, ensuring that the recommendation process is understandable. Once the Random Forest model produces its decision regarding whether a method should undergo Extract Method refactoring, the prediction and the corresponding method metrics in CSV format are passed to the *Consensual Explainer Component*.

The *Consensual Explainer* uses a consensus-based strategy that integrates the outputs of three explainability techniques: SHAP, LIME, and ANCHOR. By combining the perspectives of multiple explainers, the component generates a consensual explanation, reducing the potential biases and limitations of individual methods and increasing confidence in the explanation. The result of this process is a ranked list of the five most influential features that contributed to refactoring recommendations.

It is important to note that the generation of the explanation in this step directly constitutes the *Why* criterion, which is a key component for producing a complete and understandable recommendation. This understanding is essential because it allows users to trust the automated recommendation and can support a higher acceptance rate.

K.4 Selecting the Best Fragment

In this sub-step, for methods identified as candidates for refactoring, the JSON file containing all fragments of the method is passed to the fine-tuned CodeBERT model for evaluation and ranking. This model produces a score called *affinity* for each fragment, reflecting its suitability for extraction based on the learned semantic relationships between the fragment and the rest of the method. Fragments with higher *affinity* values are considered better candidates for extraction.

The strategy for selecting the optimal fragment proceeds in two stages. First, the two fragments with the highest scores according to CodeBERT are identified. Second, among these top candidates, the fragment with the lowest *totalUsageOutsideSlice* value is selected. This metric estimates the degree of coupling between the fragment and the

remainder of the method. By minimizing this value, the risk of disrupting the original control flow is reduced, preserving the structural integrity of the method.

This selection strategy was adopted because the numerical difference between the top-scoring fragments is typically negligible, making the *totalUsageOutsideSlice* a decisive criterion for minimizing coupling. The output of this step is the fragment considered most suitable for extraction, which, together with the output of K.2, constitutes the *Where* criterion of the complete recommendation.

K.5 Generating the Recommendation

In the final step, a carefully designed prompt (Listing 6.4) was submitted to *GPT-4* via the OpenAI API to generate Extract Method recommendations. The prompt provides contextual information on the recommendation and artifacts produced in previous steps: the best fragment and its score from Step K.4 ; the top-5 ranking features generated by Consensual Explanation from Step K.3. Always follows the *W3B* criteria. By integrating these elements, the prompt ensures that the model has sufficient context to ensemble and generate the recommendations.

For the formulation of the prompt, a Zero-Shot prompting approach was chosen, in which the task is directly communicated to GPT-4 without prior training or explicit examples. This approach leverages the model's capacity for complex reasoning and contextual understanding, enabling it to generate structured technical explanations and refactoring suggestions without additional fine-tuning. Moreover, the strategy reduces the risk of introducing spurious correlations and allows for modularity and flexibility, since prompts can be easily refined or adjusted independently of the base model (ACHIAM et al., 2023).

```
prompt = f"""
You are an experienced software engineer. Your task is to help a junior developer understand why the code fragment below should be refactored using the Extract Method technique.

## Context:
The pre-trained CodeBERT model (fine-tuned for refactoring detection) analyzed a complete method and one of its extracted code fragments. It assigned a refactoring score and a normalized probability (between 0 and 1) indicating the likelihood that the fragment is a good candidate for extraction.

The motivation for applying Extract Method usually falls into one or more of the following four categories:
1. Remove Duplication
2. Improve Code Organization
3. Improve Testability
4. Reduce Method Size

Here is the input data in dictionary format:
{json.dumps(fragmento_dict, indent=2)}

## Task:
Based on the provided data, respond to the following:

1. **Model Decision Explanation**: In a single paragraph, explain why the model recommended refactoring this fragment. Use the provided probability value (in percentage) and the vaues of the features from the top-5 ranking to support your explanation, highlighting possible correlations between the feature(from the ranking) and the model's decision. Important do not use the value of  item. percentage nor item.importance_value from the explicacao_ranking.

2. **Benefits of Extraction**: In a single paragraph, discuss the key reasons this fragment should be extracted, citing one of the four categories above.

3. **Suggested Refactoring**:
    - **Method Name**: Propose a clear and appropriate name for the new method (following naming best practices).
    - **Refactored Code**: Provide a code example showing how the original method would look after extracting the new method, including both the updated method and the new method definition. Important, format the Java code with proper line breaks and indentation. Do not write the code on a single line. It must be clean and readable, like source code in an IDE.

**Important:** Provide only the three sections above, without introductory phrases like "Certainly" or "Let's begin".
"""
```

Listing 6.4 – Prompt sent to GPT-4

To ensure consistency and clarity in the model's responses, the prompt was carefully structured into four components, described as follows.

1. **Role (Lines 2)**
    - **Description:** Defines the perspective the model should assume.
    - **Content:** You are an experienced software engineer. Your task is to help a junior developer understand why the code fragment below should be refactored using the Extract Method refactoring.
2. **Context (Lines 5)**
    - **Description:** Provides context about the approach and the analyzed code fragment, helping the model understand the purpose of the refactoring.
    - **Content:** The pre-trained CodeBERT model (fine-tuned for refactoring detection) analyzed a complete method and one of its extracted code fragments. It assigned a refactoring score and a normalized probability (between 0 and 1) indicating the likelihood that the fragment is a good candidate for extraction.
3. **Input / Information Provided (Lines 7–14)**
    - **Description:** Lists the data provided to the model for analysis before performing the tasks.
    - **Content:**
        - Four general motivations for applying refactoring, derived from the analyzed instances and the list of motivation identified by Henriques et al. (HENRIQUE et al., 2021) (Lines 7–11)
        - Result of previously executed sub-tasks: Method name, fragment, score, Top-5 ranking (Lines 13–14)
4. **Tasks (Lines 17–27)**
    - **Description:** Specifies the instructions the model should follow and the expected outputs. Each item corresponds to a section the model must produce.
    - **Content:**

a) **Model Decision Explanation:** Explain in natural language why the model recommends refactoring the fragment, supported by the top-5 ranked features.

b) **Benefits of Extraction:** Formulate the benefits of applying the refactoring, citing one of the four general motivations.

c) **Suggested Refactoring:** Propose how the refactoring should be applied, including a suggested name for the new method and an example of the refactored code. The Java code must be properly formatted with line breaks and indentation, ensuring readability.

6.3 Final Considerations

This chapter presented the proposed approach for generating the Refactoring Recommendation approach, structured through a modular pipeline that integrates four complementary phases described in detail throughout this work.

The proposed strategy differs from the traditional recommendation approach by incorporating four interconnected dimensions: (i) a specialized extractor of metrics and fragments, (ii) a Random Forest model capable of identifying opportunities for Extract Method refactoring, (iii) an explainability module that enhances the transparency and interpretability of the recommendations, and (iv) a pre-trained CodeBERT model responsible for identifying the most suitable code fragment candidate.

The integration of these components culminates in the generation of an approach that generates *Complete Extract Method Recommendations*, grounded in real refactorings and leveraging the semantic representation of the code.

An important contribution of this approach lies in its modularity and extensibility. Each component of the pipeline can be independently refined or replaced without jeopardizing the overall system architecture. This design reinforces the robustness of the solution and enables scalability, ensuring that the framework can accommodate advances in software engineering practices and machine learning models.

Nevertheless, some limitations must be acknowledged. The effectiveness of the system is inherently influenced by the quality of the training data, the characteristics of

the projects under analysis, and the constraints of the identification of the fragments.

These aspects open avenues for future research. Promising directions include the incorporation of more sophisticated fragment identification, the integration of different large language models (LLM) to enhance contextual reasoning, and the exploration of alternative XAI techniques.

Chapter 7
THE REFACTORING RECOMMENDATION API

7.1 Initial Considerations

This chapter presents the software API developed to support *Complete* Extract Method Recommendations. The solution is designed as an API to allow future integration with an IDE. For evaluation purposes, it currently provides a web interface. The description of the solution follows best practices from IEEE 1016 (Software Design Description), adapted to ensure a structured and comprehensive presentation covering functional and non-functional requirements, architecture, implementation, and user interface.

7.2 Overview of the API

The Refactoring Recommendation solution was designed primarily as a **RESTful API**, intended for integration with development tools, automated pipelines, or IDEs to support systematic analysis of Java methods. Its main objective is to provide *complete* recommendations for *Extract Method* refactoring.

Technically, the API is implemented in **FastAPI** and supports **asynchronous execution** with optional **GPU acceleration** (via PyTorch), enabling efficient processing and inference with transformer-based models such as CodeBERT and GPT-based models.

Although the core functionality is exposed through the API, a web interface (HTML + Jinja2) has been developed for demonstration and evaluation purposes. This interface allows users to interact with the system manually, providing an accessible way

to explore the tool's capabilities. It is important to note that the interface does not add new functionality; it is simply a visualization layer over the API endpoints.

Figure 11 illustrates the developer interaction with the interface. Developers provide two parameters: 1) the GitHub repository URL pointing to the class containing the method, and 2) the name of the method. The tool then analyzes the method, evaluates potential fragments for extraction, and provides comprehensive and complete recommendations.

The Extract Method Refactoring Recommendation tool is publicly available in the replication repository (ANGULO, 2025a).

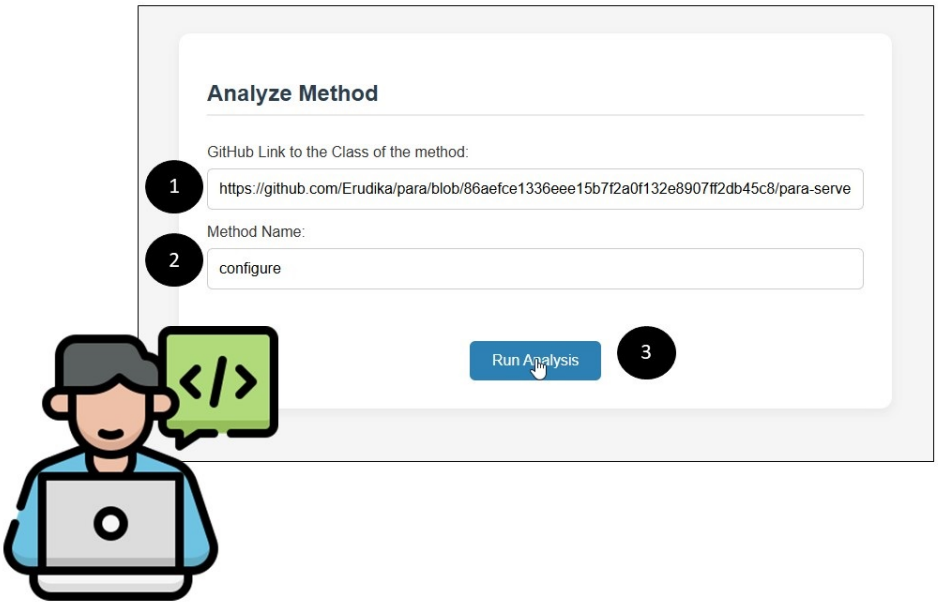


Figure 11 – Overview of tool usage

7.3 Use Cases

The Refactoring Recommendation Tool is primarily designed as a **RESTful API**, although a web interface is provided for demonstration and evaluation. The use cases focus on how clients, either developers using the web interface or external tools, interact with the API to obtain Extract Method refactoring recommendations.

Figure 12 presents the corresponding Use Case Diagram. The two main use cases are:

- **UC1: Submit Method for Analysis** In this use case, the client provides the GitHub repository URL and the name of the Java method to be analyzed. Currently, this input can be submitted through the web form for evaluation purposes. For programmatic clients, the API also supports receiving the same information as a JSON payload via an HTTP POST request. The API then orchestrates the internal processing pipeline.
- **UC2: Receive Refactoring Recommendation** After completing the analysis, the client receives a structured response. When using the web interface, the response is rendered in HTML for evaluation. For programmatic access, the API returns a JSON response containing ranked candidate fragments, associated refactoring scores, probabilities, and natural language explanations, both individual and consensus-based. External tools, such as IDE plugins or automated pipelines, can consume this information to support automated extract method recommendations.

These use cases illustrate the core workflow of the API: from the submission of a method for analysis to the retrieval of complete Extract Method refactoring recommendations. Functional and non-functional requirements associated with these interactions are detailed in Appendix A.

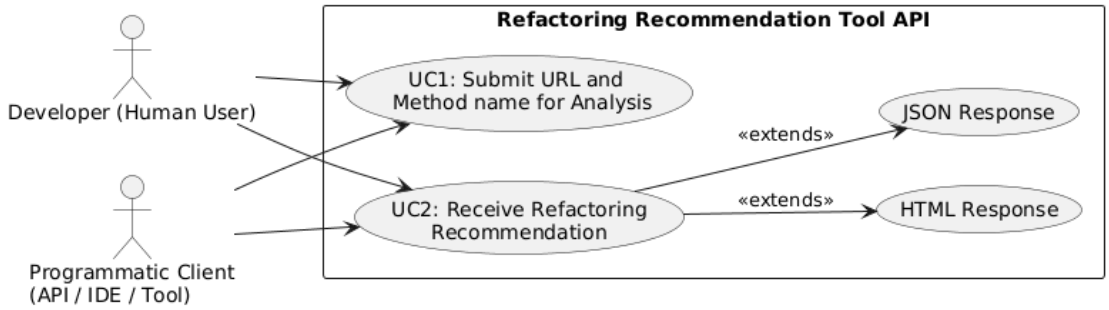


Figure 12 – Use case Diagram - Refactoring Recommendation Tool

7.3.1 Interface

7.3.1.1 User Interface

The User interface of the Refactoring Recommendation solution provides an intuitive and structured way for developers to interact with the system. Figure 13 shows a screenshot of the form.html page, highlighting its layout, input fields, and submission button. The form enables the user to specify the URL of the GitHub repository (linking directly to the class containing the method) and to enter the name of the method to be analyzed.

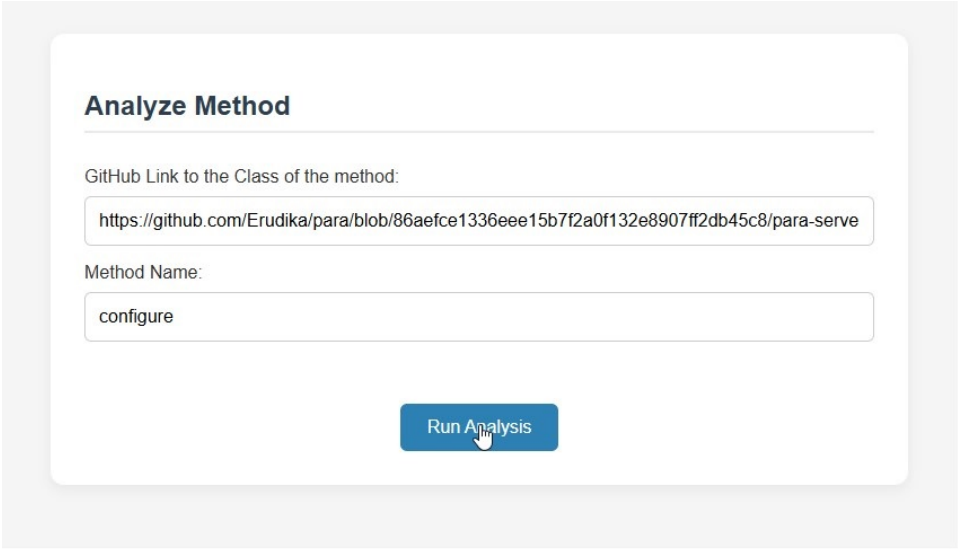


Figure 13 – User Interface - Input values

Figure 14 shows a screenshot of the *result.html* page, which presents the outcomes of the analysis. The page is structured into three main sections, each focusing on a specific aspect of the recommendation.

The first section, labeled as **I. Recommendation** focuses on the *Which* and *Where* criteria. This section displays the GitHub repository URL, class name, method name, the refactoring probability, which is the score calculated by the fine-tuned CodeBERT model, and the selected fragment for extraction.

Extract Method Refactoring Recommendation Results

I. Recommendation

GitHub Repository URL: https://github.com/Erudika/para/blob/86aefce1336eee15b7f2a0f132e8907ff2db45c8/para-server/src/main/java/com/erudika/para/search/SearchModule.java

Class Name: SearchModule.java

Method Name: configure

Refactoring Probability: 74.01%

Selected Fragment:

```
bind(Search.class).to(LuceneSearch.class).asEagerSingleton();
```

II. Explanation based on Features(Ranking)

Based on a qualitative analysis, the following 5 metrics played a key role in the decision to recommend this method for refactoring. They are listed below in descending order of importance — from the most to the least influential.

Feature	Description	Value
methodRfc	Response For a Class (adapted to method) – number of methods that could be executed in response to a call to this method	8
methodMaxNestedBlocks	Maximum depth of nested code blocks in the method	3
methodParametersQty	Number of parameters declared in the method	0
methodLoc	Lines of Code	20
methodCbo	Coupling Between Object classes in the method	3

III. Application

1. **Model Decision Explanation**: The model recommended refactoring this fragment with a probability of 74.01%, indicating a strong likelihood that the fragment is a good candidate for extraction. This decision is influenced by the method's complexity and size, as evidenced by the feature values. The method has a Response For a Class (RFC) value of 8, suggesting multiple methods could be triggered, which can increase complexity. The maximum nested block depth is 3, indicating a relatively deep nesting that can complicate understanding and maintenance. The method has 20 lines of code, which, while not excessive, suggests that breaking it down could improve readability. These factors, combined with the fragment's repetition within nested conditions, support the decision to extract the fragment to improve the method's structure and maintainability.
2. **Benefits of Extraction**: Extracting the fragment "bind(Search.class).to(LuceneSearch.class).asEagerSingleton();" will primarily improve code organization. By extracting this repeated binding logic into a separate method, the code becomes more modular and easier to understand, which simplifies maintenance and reduces the potential for errors. Additionally, having a dedicated method for this binding logic can improve testability, as it allows for targeted testing of this specific functionality without needing to execute the broader method context.
3. **Suggested Refactoring**:
    - **Method Name**: bindLuceneSearchAsSingleton
    - **Refactored Code**:

```
protected void configure() {
        String selectedSearch = Config.getConfigParam("search", "");
        if (StringUtils.isBlank(selectedSearch)) {
            bindLuceneSearchAsSingleton();
        } else {
            if ("lucene".equalsIgnoreCase(selectedSearch)) {
                bindLuceneSearchAsSingleton();
            } else {
                Search searchPlugin = loadExternalSearch(selectedSearch);
                if (searchPlugin != null) {
                    bind(Search.class).to(searchPlugin.getClass()).asEagerSingleton();
                } else {
                    // default fallback - not implemented!
                    bind(Search.class).to(MockSearch.class).asEagerSingleton();
                }
            }
        }
    }

    private void bindLuceneSearchAsSingleton() {
        bind(Search.class).to(LuceneSearch.class).asEagerSingleton();
    }
```

Figure 14 – Complete Extract Method Refactoring Recommendation

The second section, labeled **II. Explanation Based on Features (Ranking)**, focuses on the *Why* criterion and presents the consensual explanation based on the top five features that most influenced the model's recommendation. This ranking provides a rationale for *why* the model suggests refactoring the method. Each feature is accompanied by its name, a brief description, and its corresponding value in the analyzed code, helping users understand the factors driving the refactoring decision.

The third section, labeled **III. Application**, is divided into three subsections:

1. **Model Decision Explanation – Focus on the Model.** Explains, in natural language, *why* the model decided that the method and the selected fragment should be refactored. This explanation combines the feature ranking (Top-5 features) with the predicted refactoring probability, highlighting possible correlations between the features and the model's decision. It helps users understand the rationale behind the recommendation and the relative importance of each feature in influencing the model.

2. **Benefits of the Extraction – Focus on the Fragment.** Describes the main benefits of extracting the selected fragment. This explanation draws from a generic catalog of refactoring motivations, contextualized for the analyzed code. It provides insight into how the extraction can improve code readability, maintainability, and modularity, supporting better design decisions.

3. **Suggested Refactoring.** Outlines how the refactoring can be applied in practice, including proposing a new method name, showing the refactored method, and presenting the newly created method containing the extracted content. This section gives insights of how to implement the recommendation, ensuring practical usability.

7.3.1.2 Programmatic Interface

The Refactoring Recommendation API provides a programmatic interface designed for integration with external tools, IDE plugins, or automated pipelines. This interface exposes the same core functionality as the web-based evaluation, but is accessed via HTTP requests and responses in JSON format.

The API provides dedicated endpoints for programmatic consumption by external systems. The primary endpoint is:

```
{
"repo_url": "https://github.com/user/repo",
"method_name": "Adittion"
}
```

Listing 7.1 – POST /analyze – Submit Java Method for Analysis

The response returns the analysis results, including the best candidate fragment, refactoring score, probability, and natural language explanations:

```
{
 "Method Name": "configure",
"probability": "87%",
"fragment": "System.out.println(x);",
"explicacao_ranking": "Consensus explanation combining top features...",
"explicacao": "Ensamble explanation generated by ChatGPT..."
}
```

Listing 7.2 – JSON Response for Method Analysis

The API uses standard HTTP over REST, allowing external tools to interact programmatically. Requests must include the following headers:

```
Content-Type: application/json
Accept: application/json
```

Clients can submit repository URLs and method names via POST requests, receive structured JSON responses, and integrate the recommendations directly into automated refactoring pipelines or IDE plugins. Asynchronous execution and GPU acceleration ensure efficient processing of multiple requests.

7.4 Architecture and Design

The Refactoring Recommendation Tool is implemented as a centralized API, with *FastAPI* serving as the core orchestrator. It exposes endpoints that accept input parameters (GitHub URL and method name) and return structured recommendations. The system offers two complementary modes of access: (i) a web-based interface designed for evaluation and manual testing, and (ii) a programmatic interface designed for integration with external systems, IDE plugins, or automated pipelines.

The architecture is designed to integrate multiple specialized components in a modular, layered manner, as illustrated in Figure 15. It is organized into three main layers: *Interface Layer*, *Processing Layer*, and *External Services*. This separation of concerns allows efficient coordination of data flow between user inputs, internal analysis modules, and external services, while promoting maintainability and extensibility.

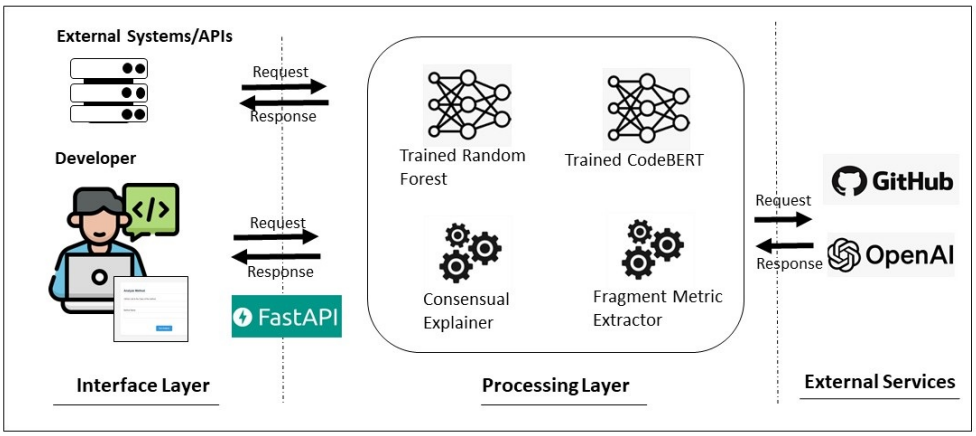


Figure 15 – Overview of the Architecture

7.4.1 Architecture Layers

Figure 16 shows the component diagram, detailing the three main layers and their interactions:

The Interface layer. For this layer, we have the User interface and the Programmatic Interface. The User Interface is implemented as a form-based frontend (`form.html`) that allows users to provide the GitHub repository URL and the target method name as input parameters. Once submitted, the backend processes the request and renders the results as an HTML page (`result.html`). This output includes the three sections explained in 7.3.1.1 . The interface thus enables non-technical users and evaluators to interact with the system intuitively, without requiring knowledge of API calls or JSON data structures, making it particularly suitable for manual analysis and evaluation purposes.

On the other hand, the Programmatic Interface is exposed through the dedicated `POST /analyze` endpoint, which accepts JSON input containing the fields `repo_url` and `method_name`. This endpoint is designed for automated consumption by other APIs, IDE plugins, or software engineering tools, enabling seamless integration into larger workflows. Clients must include the header `Content-Type: application/json` when submitting requests, and the response is returned in structured JSON format. The output includes the elements described in 7.3.1.2.

The Processing Layer. This layer encapsulates core internal components responsible for analyzing source code and generating the extract method refactoring recommendation. This layer includes:

- *Fragment Metric Extractor* (Java JAR): Parses the source code to extract fragments and calculates structural and behavioral complexity metrics;
- *Random Forest Classifier*: Identifies methods as candidates for refactoring based on the extracted metrics;
- *CodeBERT Predictor*. Evaluates the semantic similarity of code fragments to choose the most suitable one;
- *Consensus Explainer module*: Determines the most influential metrics for each prediction and presents the top five in a ranked list, improving interpretability.

The *FastAPI backend* aggregates the outputs from the processing modules and invokes the *OpenAI GPT API*, an external API service, to assemble and refine the complete recommendation into clear, developer-friendly natural language.

The External Services layer. This layer encompasses external resources that the tool relies on for its operation. This layer includes the *GitHub Repository*, which serves as the source of Java code to be analyzed, providing the raw methods for processing. It also includes the *OpenAI GPT API*, an external service used to assemble and refine the complete refactoring recommendations into clear, developer-friendly natural language. By separating these external dependencies, the layer highlights that these services are accessed remotely and are essential for overall system functionality.

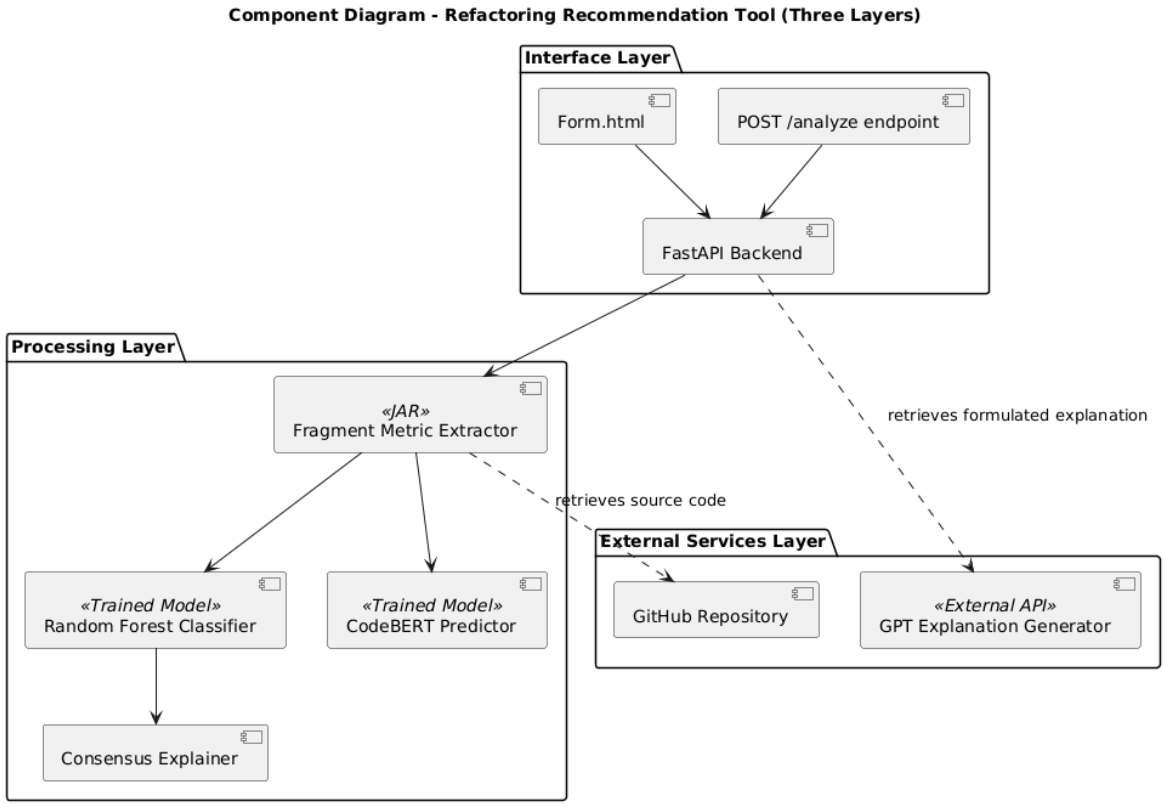


Figure 16 – Component Diagram - Refactoring Recommendation Tool

7.4.2 Data Flow

This subsection shows the sequence diagrams that describe the interaction flow between the Refactoring Recommendation Tool components during the processing of a request, either via the web interface or programmatically via the JSON API.

User Interface, the Figure 17 presents the sequence diagram for the user workflow. The flow begins when the user submits, through the web interface form, the URL of a GitHub repository and the name of the method to be analyzed. Then, the *FastAPI Backend* receives this request and triggers the *FME* (Fragment Metric Extractor) module, implemented as a Java JAR, responsible for downloading the repository's source code from the *GitHub* repository and extracting relevant fragments and metrics. As output, this module returns JSON and CSV files containing the processed information.

The *FastAPI* backend then loads the *Random Forest* Model and calculates the probability that the method is a candidate for refactoring using Extract Method. If this probability exceeds the 0.50 threshold, the backend triggers the *Consensual Explainer*, which generates an explanation (ranking) based on the most influential metrics.

Based on this decision, the *FastAPI* invokes the *CodeBERT* Model, which identifies the most suitable code fragment for refactoring and returns additional information, such as score and probability. These results, along with the metric ranking, are sent to the external *GPT API service*, which ensembles and refines the complete recommendation. Finally, the *FastAPI* aggregates all the results and returns a response to the user containing the calculated probability, the suggested fragment, the technical explanation (metric ranking), and the natural language explanation generated by GPT (Model decision explanation, Benefits, and Suggested refactoring application).

Programmatic Interface. Figure 18 presents the sequence diagram for the programmatic workflow. The flow begins when an external client submits a POST request to the `/analyze` endpoint with a JSON payload containing the fields `repo_url` and `method_name`. The internal processing pipeline is identical, involving FME, Random Forest, CodeBERT, and GPT modules.

Finally, *FastAPI* aggregates all results and returns a structured JSON response to the client. The output contains the method name, probability, suggested fragment, technical explanation (ranking), and GPT-generated natural language explanation, including model rationale, benefits, and suggested refactoring application as described in 7.3.1.2.

This dual flow ensures that both human users and external systems can interact seamlessly with the Refactoring Recommendation Tool, either for manual evaluation or automated integration, without requiring knowledge of the internal processing pipeline.

7.4.3 System Deployment

The deployment diagram is illustrated in Figure 19, which shows the physical distribution of the Refactoring Recommendation Tool components and their communication channels.

The entire application stack is deployed on a *local host*, which runs all core modules and orchestrates their execution. The *FastAPI Backend* operates as the central

controller for both the User Interface and the Programmatic Interface, coordinating the invocation of the *Random Forest Classifier*, *CodeBERT Model*, *Consensus Explainer*, and the *Fragment Metric Extractor*.

The *Random Forest Classifier* and *CodeBERT Model* are pre-trained models loaded in memory at runtime, receiving input data from the backend via internal Python calls. The *Consensus Explainer* is also invoked through internal Python calls. The *Fragment Metric Extractor*, implemented as a Java JAR, is executed from the backend using command-line invocation (`java -jar`), ensuring interoperability between Python and Java components.

Both interfaces share this same deployment configuration:

- **User Interface:** Input parameters are submitted through the form-based frontend (`form.html`), and results are rendered as HTML pages (`result.html`). This allows non-technical users to interact with the system intuitively for manual analysis and evaluation.
- **Programmatic Interface:** Input parameters are submitted as JSON payloads to the `/analyze` endpoint, and structured JSON responses are returned. This mode enables automated integration with external APIs, IDE plugins, or other software engineering tools.

The system also relies on external services over the HTTP/HTTPS protocol. The *GitHub Repository* provides access to source code artifacts, fetched by the *Fragment Metric Extractor*, and the *OpenAI GPT API* is accessed exclusively by the backend to ensemble and refine the complete recommendation. This unified deployment demonstrates that the tool can support multiple consumption modes without duplicating infrastructure, ensuring maintainability and efficient resource usage.

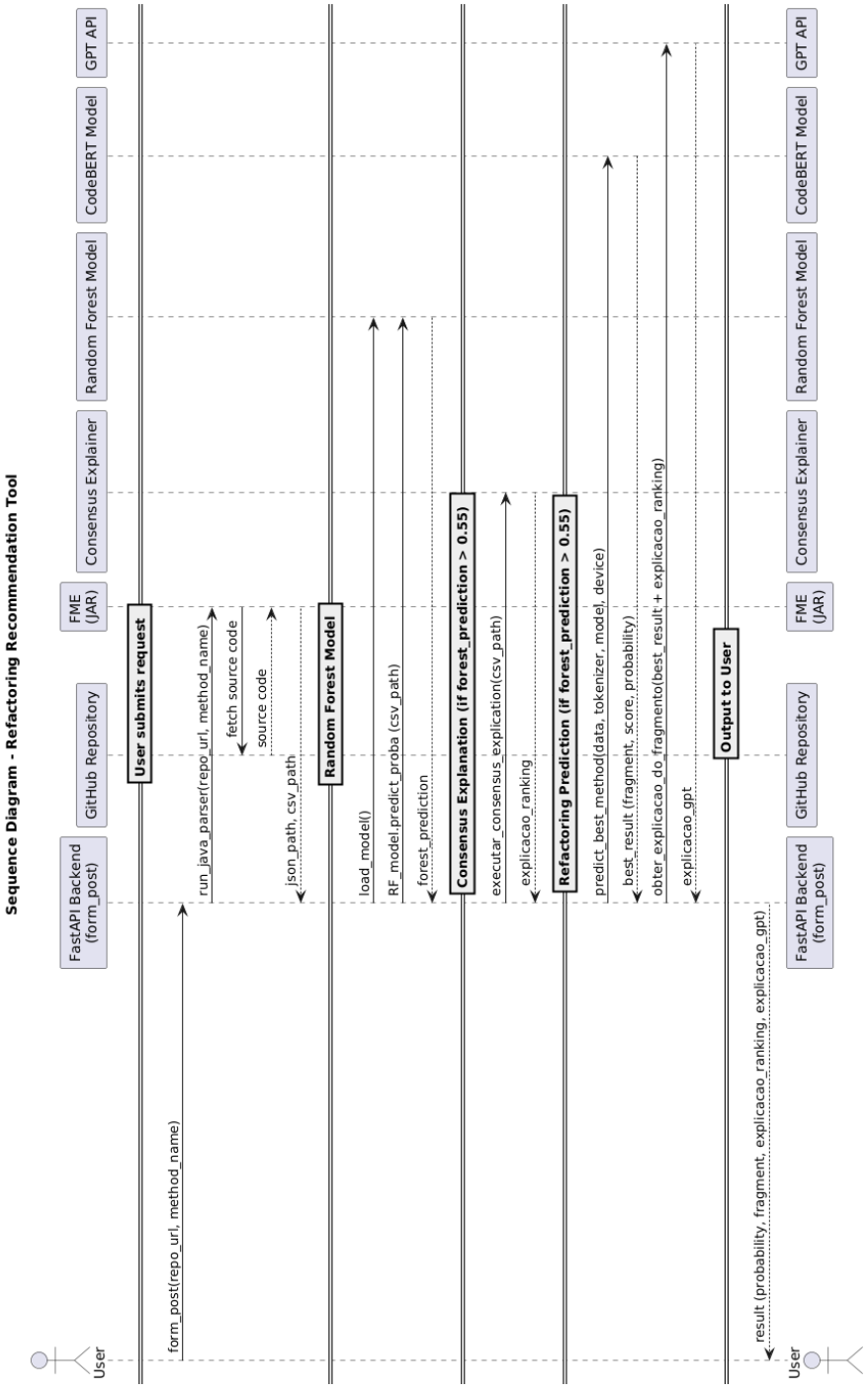


Figure 17 – Sequence Diagram - Refactoring Recommendation Tool

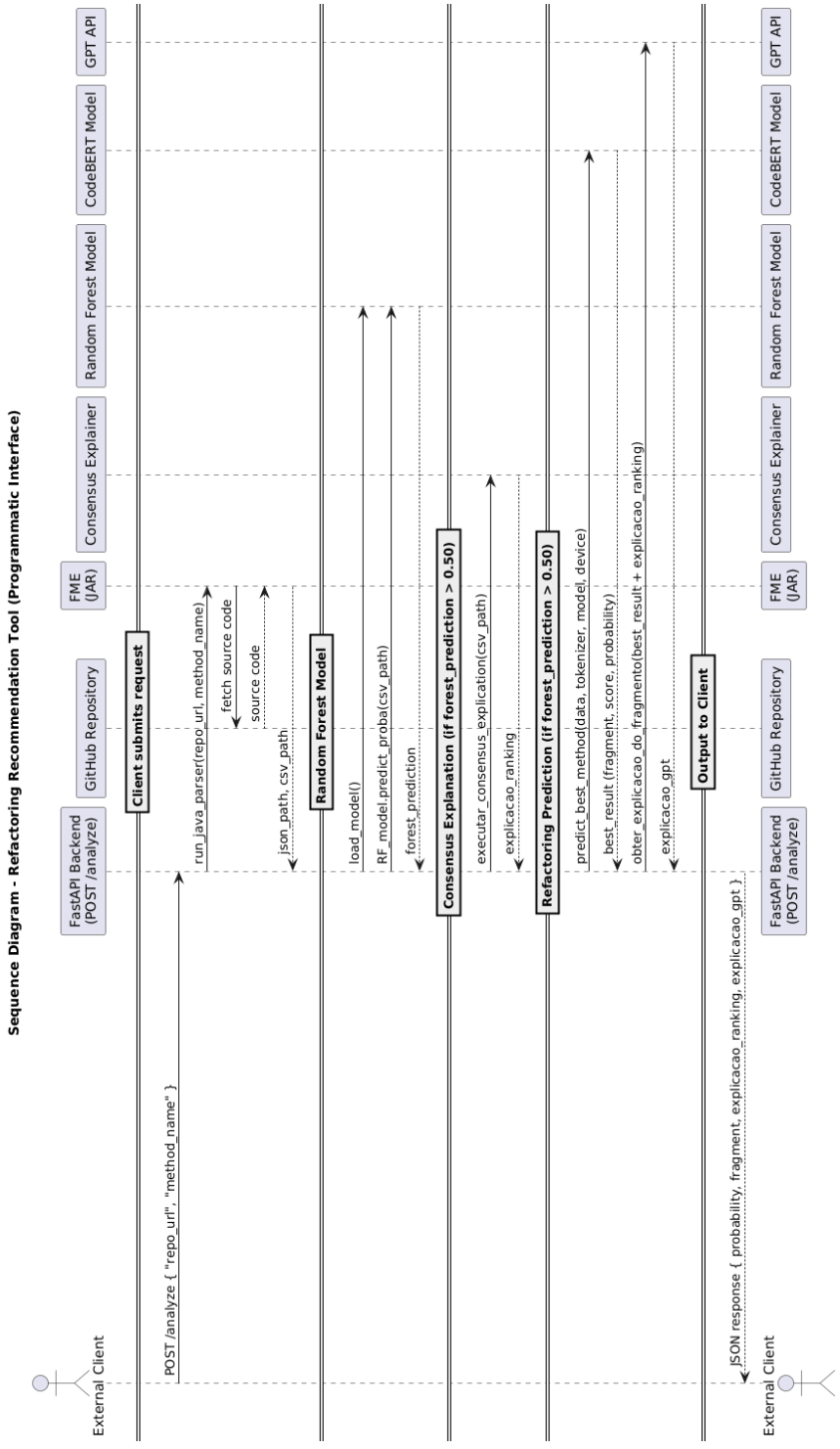


Figure 18 – Sequence Diagram - Programmatic Interface

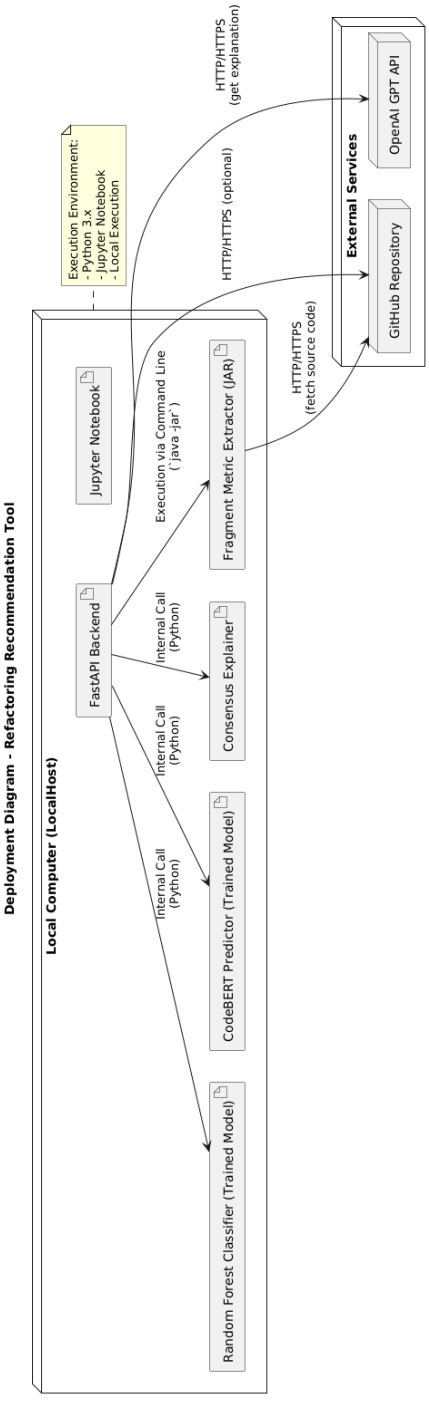


Figure 19 – Deployment Diagram - Refactoring Recommendation Tool

7.4.4 Design Considerations

The Refactoring Recommendation Tool is designed for flexibility and adaptability. The fine-tuned CodeBERT model, for instance, can be updated to a newer version after retraining, requiring only a straightforward substitution without impacting the overall processing pipeline. Similarly, the Random Forest classifier used for method classification supports the integration of alternative or additional machine learning models with minimal modifications to existing components.

The Consensual Explainer is structured as a modular and extensible component: while it currently combines three baseline techniques (SHAP, LIME, and ANCHOR), it can be easily extended to incorporate new explainability methods as they become available. Likewise, the Java-based Fragment Metric Extractor (FME) can be replaced with an improved version while preserving full compatibility with the rest of the system.

This modular and interchangeable architecture ensures that the tool can evolve with advances in machine learning, code analysis techniques, or interpretability methods, minimizing the need for extensive re-engineering and promoting long-term maintainability.

7.5 Final Considerations and Future Work

This chapter presented the Refactoring Recommendation solution, detailing its architecture, processing pipeline, and user interfaces. While the tool currently provides a web-based interface for demonstration and evaluation, it was designed from the ground up as a programmatic API, enabling integration into IDEs and automated software engineering workflows.

Future work includes deploying the tool as a publicly accessible web service, optimizing model loading and execution to reduce response latency, and expanding support to additional programming languages. Further enhancements could involve integrating more advanced machine learning models, incorporating new explainability techniques, and improving the ensemble of recommendations. These developments aim to increase the tool's accessibility, performance, and versatility, ensuring it remains adaptable to diverse developer needs and emerging technologies.

Chapter 8
EVALUATION

8.1 Initial Considerations

This chapter presents a controlled experiment designed to focus on understanding how the complete recommendation influences developers' perceptions, confidence, and acceptance of the recommendations. Postgraduate students analyzed Java methods under two conditions: without tool support (baseline) and with tool support. The following sections detail the controlled experiment.

8.2 Controlled Experiment

The goal of this experiment is to provide evidence of whether the complete refactoring recommendations contribute to a better understanding of the recommendation, increase user confidence, and improve the acceptance of suggested refactorings. To do that, we conducted a controlled experiment and designed a 20-question questionnaire. The experiment compares two conditions: (*i*) a baseline scenario, referred to as *without tool support*, in which participants receive only a basic recommendation; and (*ii*) an enhanced scenario, referred to as *with tool support*, in which participants receive a complete refactoring recommendation.

Figure 20 follows the experimental process proposed by (WOHLIN, 2000), which consists of the following steps: scoping, setting, planning, and results analysis. These steps are described in detail below.

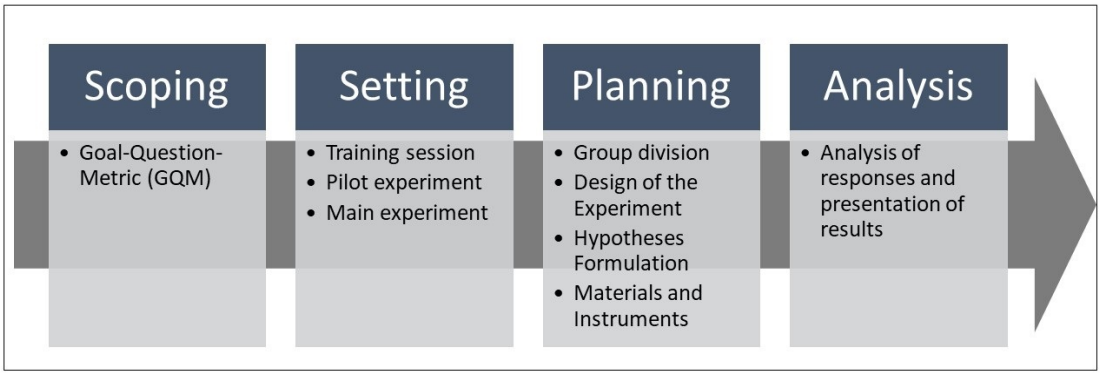


Figure 20 – Experiment Process

8.2.1 Scoping

One of the expected benefits of using the refactoring recommendation tool is obtaining a complete refactoring recommendation, which can enhance the understanding of the recommendation, increase the user's confidence in it, and ultimately improve the acceptance rate of refactoring suggestions.

Following the Goal–Question–Metric (GQM) (WOHLIN, 2000), the scope of this experiment can be expressed as:

Analyze the complete refactoring recommendations
for the purpose of evaluating its usefulness in terms of user perception
with respect to the understanding, confidence, and acceptance of refactoring recommendations
from the point of view of researchers conducting empirical studies in software engineering
in the context of postgraduate computer science students performing refactoring tasks during a controlled classroom session.

8.2.2 Setting

The experiment was conducted during two class sessions in July 2025. The participants were seven postgraduate students in the course *Object-Oriented Software Development*. The experiment was structured into three main activities:

1. **Training session:** This session covered the main theoretical topics, including Refactoring, Code Smells, Extract Method Refactoring, Structural and Behavioral Metrics, and Explainable AI (XAI).

2. **Pilot experiment:** A pilot study was conducted to familiarize the participants with the experimental procedure, the questionnaire format, and the use of the refactoring recommendation tool. Based on the pilot results, we analyzed the following aspects: the average time required to answer the Google form, the clarity of the instructions, and the accessibility of the tool URL. In addition, participants completed a user profile questionnaire to assess their level of experience in Java programming, which allowed us to categorize them into two groups. The profile form and corresponding results are available in Appendix B.

3. **Main experiment:** In the first part, participants signed a virtual consent form indicating that the collected data would be used solely for academic and scientific purposes. The experiment then proceeded in the same format as the pilot study but involved different Java methods for analysis.

8.2.3 Planning

8.2.3.1 Participants and Group Division

Initially, nine participants completed the profile form, but only seven effectively took part in the experiment. Participants were divided into two groups according to their level of expertise in Java programming, as the experimental tasks involved analyzing and evaluating Java methods. Familiarity with Java is critical to ensure that participants can accurately understand code structures, providing meaningful feedback about recommendations. The profile form and corresponding results are available in Appendix B, as well as in (ANGULO, 2025b).

As a result of this division, we obtained **Group G01**, comprising three parti-

cipants with higher expertise in Java programming, and **Group G02**, comprising four participants with medium-to-low expertise. This grouping allowed us to analyze how the level of expertise influences the perception and acceptance of refactoring recommendations.

8.2.3.2 Design of the Experiment

The experiment was structured in two phases, as shown in 21. In the first phase, participants worked without tool support, relying only on the provided information. In the second phase, they performed the same tasks with the complete recommendation provided by the Refactoring Recommendation Tool.

Phase 1 (without tool support).

The first phase aimed to capture participants' decisions when faced with an unsupported recommendation; that is, a raw suggestion without explanatory context. To this end, participants were provided with the source code of the method, a set of structural and behavioral metrics, and additional contextual information. Since a developer's decision to apply a refactoring often depends on their knowledge of the system, particularly the classes and methods they have worked on, it was important to simulate this context. As only isolated methods were shown in the experiment, we complemented them with a textual description of the method's responsibilities and in-code comments describing blocks of logic.

This phase was divided into two parts. In the first part, participants were asked to indicate whether they would apply the refactoring, considering only the information provided (Q1). For cases where the answer to Q1 was *Yes*, Q5 required participants to manually identify the fragment (lines of code) they believed should be extracted. In the second part, the form shows the fragment recommended by the tool (still without explanation). Participants were asked to compare it with their own selection (Q5–Q09). All decisions in this phase were made solely based on the information explicitly provided.

Phase I concluded with participants reporting their overall confidence in their decision to refactor, which was captured through question Q10, measured on a Likert scale.

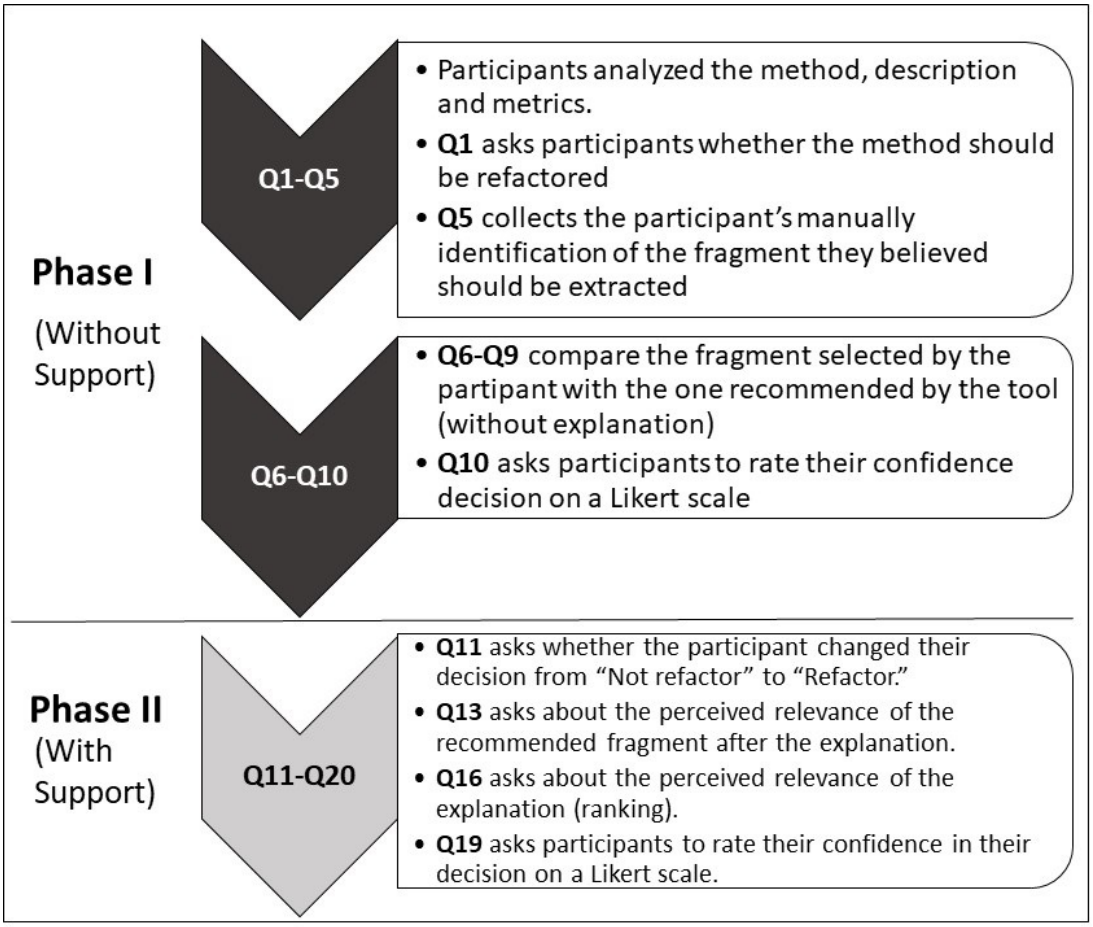


Figure 21 – Phases Process

Phase 2 (with tool support). In the second phase, with the objective of comparing the responses, the participants analyzed the same methods that were analyzed in Phase I, but this time with the support of the Refactoring Recommendation Tool, answering questions Q11 to Q20. The tool provided a complete recommendation that included the suggested fragment with its associated probability score, a ranking of the top five metrics that influenced the recommendation, a textual explanation highlighting the rationale and potential benefits of the refactoring, and a sketch of the method after applying the Extract Method refactoring.

In this phase, Q11 collected information on whether participants changed their

initial decision compared to Q1. Q13 investigated the perceived importance of the recommended fragment, while Q16 asked about participants' views on the metric ranking. Finally, Q19 captured their level of confidence when making a refactoring decision with the support of a complete recommendation.

This experimental design allows for a systematic comparison between unsupported and tool-supported scenarios. It generates empirical evidence on how complete recommendations affect developers' acceptance, confidence, and perception, thereby contributing to a broader understanding of the effectiveness and adoption of refactoring tools. The questionnaire was implemented in Google Forms, which facilitated both distribution to participants and the systematic collection of responses. The full form is available in (ANGULO, 2025b).

8.2.3.3 Hypotheses Formulation

In alignment with the defined scope, this study pursues two main research objectives. First, to investigate whether using the tool increases users' confidence in the refactoring recommendation. Second, to assess whether providing a complete recommendation leads to a higher acceptance rate of the recommendation compared to scenarios without tool support. Together, these objectives allow for a clear assessment of how automated tool support affects developers' decisions during refactoring tasks.

Based on these objectives, the following hypotheses were formulated:

- **H_1:** Providing a complete recommendation supported by our tool increases the acceptance rate of refactoring suggestions compared to not having tool support.
- **H_2:** Providing a complete recommendation improves users' perception of confidence in the recommendation.

8.2.3.4 Materials and Instruments

Questionnaire. The questionnaire designed for this experiment is presented in two complementary tables for clarity. Table 16 provides the full text of all questions, organized by phase: Phase 1 (without tool support) and Phase 2 (with tool support). Conditional

questions, which are only answered depending on previous responses, are indicated in parentheses. Likert scales are specified where applicable.

Q#	Question
Phase 1 — Without Tool Support	
Q1	Analyzing the method and all the information provided, do you believe this method should undergo an Extract Method?
Q2	Please justify your answer to Q1.
Q3	(If Q1 = "Yes") Did the metrics of the code help you make your decision?
Q4	(If Q3 = "Yes") Which five metrics did you consider most important for your decision? Please rank them in order of importance, if possible.
Q5	(If Q1 = "Yes") Which code fragment (indicate starting and ending lines) do you believe should be extracted into a new method? Please justify your answer.
Q6	(If Q5 was answered) Is the code fragment recommended by the tool the same as the one you provided in Q5?
Q7	(If Q6 = "No") Do you think the code fragment you chose is as relevant as the one suggested by the tool?
Q8	(If Q1 = "No"or "I'm not sure / Can't decide") Now that you know the tool recommended refactoring this method and you have seen the suggested code fragment, do you consider this code fragment a good candidate for extraction?
Q9	Please justify your answer to Q8.
Q10	How confident were you in your refactoring decision? (Scale: 1 = Not confident at all, 5 = Very confident)
Phase 2 — With Tool Support	
Q11	(If Q8 = "No"or "I'm not sure / Can't decide") Please analyze the sections "Explanation based on Features (Ranking)" and "Model Decision Explanation". Now that you have this information, would you change your mind? Do you think it makes sense to refactor this method?
Q12	Please justify your answer to Q11.
Q13	(If a code fragment was provided in Q5, and Q7 = "No"or "I'm not sure / Can't decide") Does the information in "Feature-based Explanation (Ranking)" and "Explanation of the Model's Decision" lead you to reconsider your choice and regard the tool-selected fragment as more relevant?
Q14	Please justify your answer to Q13.
Q15	Did the ranking metrics contribute to your decision-making process in Q11 and/or Q13?
Q16	Do you agree with the ranking shown in "Explanation based on Features (Ranking)"?
Q17	Do you agree with the benefits outlined in the "Benefits of Extraction" section?
Q18	Please justify your answer to Q17.
Q19	Considering all explainability elements — probability, ranking, explanation of benefits, and refactoring attempt — how confident are you in your refactoring decision? (Scale: 1 = Not confident at all, 5 = Very confident)
Q20	Did all the elements (sections) shown by the tool help you better understand or feel more confident about the recommendation? (Scale: 1 = Not at all, 5 = Very much)

Table 16 – Evaluation questions used in both phases of the experiment.

For a compact overview, Table 25 in Appendix B summarizes the same questions, showing only the question ID, type (Closed-ended, Open-ended, or Likert), and conditional dependencies. The questionnaire was implemented in Google Forms, which facilitated both distribution to participants and systematic collection of responses. The

full form is available in (ANGULO, 2025b).

Each questionnaire included a combination of closed-ended and open-ended questions. The closed-ended items were numerically coded and comprised three types of responses: (i) binary (e.g., Yes/No decisions or agreement with specific recommendations), (ii) categorical (e.g., multiple-choice judgments on relevance), and (iii) Likert scales (ranging from low to high levels of confidence or acceptance). This structure enabled precise quantification of participants' choices, perceptions, and degree of agreement. The open-ended questions complemented these by allowing participants to justify their selections and provide qualitative feedback, offering richer insights into their reasoning.

Evaluation Dataset. The evaluation relies on a dataset of 2091 positive samples, none of which were used in the training of the Random Forest or CodeBERT models. To uncover potential patterns that might influence the quality or nature of explanations, we applied an unsupervised K-means clustering algorithm.

The clustering analysis, illustrated in Figure 22, revealed three well-defined groups of methods. The *C-high* cluster comprises 144 instances characterized by high metric values; the *C-low* cluster includes 1283 instances with low metric values; and the *C-mid* cluster contains 664 instances with medium-level metrics. Table 17 complements this visualization by presenting the mean values of the metrics across clusters, highlighting distinct method profiles: methods in *C-high* are larger and more complex, while those in *C-low* tend to be smaller and simpler.

For the user experiment, we selected eight examples: four from the *C-low* cluster and four from the *C-high* cluster. Methods were ordered by complexity (from lowest to highest), and two constraints guided the selection: (i) methods had to contain between 24 and 55 lines of code (LOC), and (ii) they had to include structures recognized by the fragment extraction tool (*FME*). The LOC restriction was applied to avoid excessively large methods that would be too time-consuming to evaluate, while the second constraint ensured compatibility with the tool. Together, these criteria enabled a representative sample of both simpler and more complex methods.

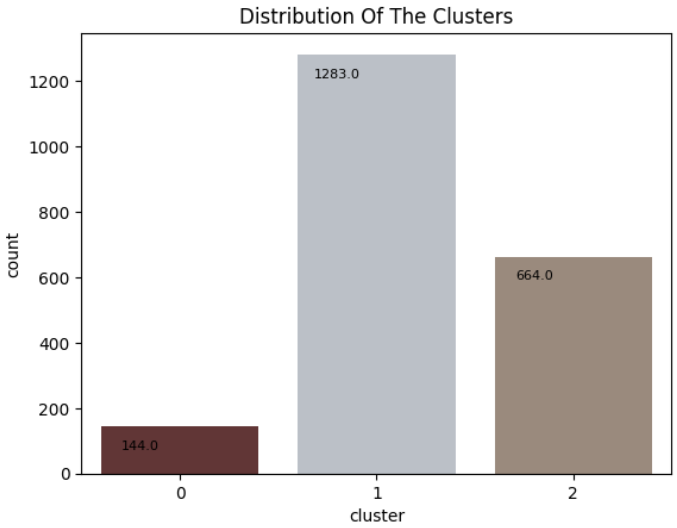


Figure 22 – Distribution of the Cluster for Evaluation

Table 17 – Clustered method metrics

Metric	C-low	C-medium	C-high
AnonymousClassesQty	0.2	0.4	0.7
AssignmentsQty	2.0	8.0	25.5
CBO	2.2	5.4	8.0
ComparisonsQty	0.2	0.9	3.4
LambdasQty	0.1	0.1	0.0
LOC	12.2	35.2	95.5
LoopQty	0.1	0.7	2.7
MathOperationsQty	0.6	1.9	9.6
MaxNestedBlocks	0.8	2.3	3.6
NumbersQty	0.8	3.0	12.8
ParametersQty	1.1	1.6	1.5
ParenthesizedExpsQty	0.2	0.5	3.0
ReturnQty	0.7	1.2	1.9
RFC	4.8	13.6	27.9
StringLiteralsQty	1.2	3.9	14.1
SubClassesQty	0.0	0.0	0.0
TryCatchQty	0.1	0.4	1.1
UniqueWordsQty	14.3	31.3	57.6
VariablesQty	1.2	5.2	16.2
WMC	2.4	6.5	20.0

Code	Method Name	LOC	Group	Cluster
G01-CB-M01	togglePatch(...)	24	G01	C-Low
G01-CB-M02	saveImage(...)	26	G01	C-Low
G01-CA-M03	drawModel(...)	32	G01	C-high
G01-CA-M04	checkProfile(...)	35	G01	C-high
G02-CB-M01	dispatchTouchEvent(...)	27	G02	C-Low
G02-CB-M02	toString()	31	G02	C-Low
G02-CA-M03	createParcel(...)	44	G02	C-high
G02-CA-M04	post()	52	G02	C-high

Table 18 – Distribution of methods across groups and metric clusters.

Finally, a set of eight evaluation forms was created, with four assigned to each participant group. Each group analyzed two methods from each cluster, as detailed in Table 18. The methods considered for this experiment are available in (ANGULO, 2025b).

8.2.4 Analysis & Discussion

The results of the analysis were grouped into the following points.

8.2.4.1 Impact of the explanations on Refactoring Decisions

To verify the decision change, we analyzed questions Q1, Q8, and Q11, which capture potential shifts in participants' opinions. In total, 26 responses were collected for Q1 (two participants analyzed only two methods). The vast majority of responses (24 out of 26) initially answered *Yes* (they agree to refactor the method), while only two participants provided a negative response (*No*); none selected *I am not sure*.

Consequently, Q8 applied exclusively to the two participants who initially answered *No*, as this question was mandatory only for those who did not select *Yes* in Q1. Similarly, Q11 was required only for participants who answered *No* or *I am not sure* in Q8. This filtering substantially reduced the number of valid cases available for analysis.

Table 19 presents the responses of the two participants who initially disagreed in Q1. Participant P01 changed their answer after viewing the fragment recommended by the tool, moving from *No* in Q1 to *Yes* in Q8, and therefore did not proceed to

Q11. In contrast, Participant P02 maintained a negative judgment throughout Q1 and Q11. Although the sample is limited, these results suggest that explanatory support may influence decision revision in some cases. However, given the very small number of applicable cases, the findings are exploratory and cannot be generalized statistically.

Table 19 – Transition of answers for participants who selected "No" in Q1.

Spreadsheet	Participant	Q1	Q8	Q11
G02-CB-M02-Experiment-EMR	P01	No	Yes	-
G01-CA-M04-Experiment-EMR	P02	No	-	No

8.2.4.2 Analysis of Recommended Code Fragments

Agreement with the Recommended Code Fragment (Q6)

In Q6, participants were asked whether the code fragment recommended by the tool corresponded to the fragment they selected themselves in Q5. Figure 23 shows the result. Out of 26 responses, 15 responses (57.7%) reported correspondence with the fragment recommended by the tool; this means that the fragment identified by the participants is the same as that identified by the tool. 10 responses (38.5%) reported non-correspondence, and one empty response (3.8%).

These results indicate that *a majority of participants (57.7%) perceived the recommended fragment as aligning with their own judgment*. This finding suggests that the tool was generally effective in providing recommendations consistent with participants' intuitive selections, supporting its potential reliability in aiding refactoring decisions.

Perceived Importance of the Recommended Fragment (Q7)

For participants whose self-identified fragment differed from the one recommended by the tool (Q6 = No). With question Q7, We evaluated the perceived importance of the fragment recommended by the tool.

Table 20 presents the results: out of 10 such cases, 5 responses (50%) considered the recommended fragment and their own selection equally relevant, 4 responses (40%) favored the fragment they had chosen, and only 1 response (10%) judged the tool's fragment to be more relevant.

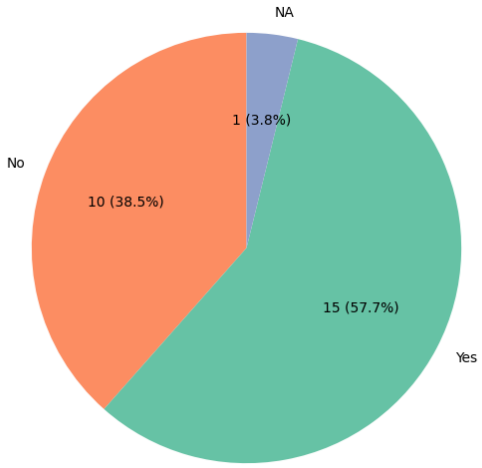


Figure 23 – Distribution of Q6 Responses

These findings suggest that even when participants initially disagreed with the recommended fragment, in half of the cases they still recognized the suggested fragment as equally important, highlighting its potential to provide meaningful fragments for Extract Method refactoring.

Table 20 – Perceived Importance of the Recommended Fragment (Q7)

Response	**Count**
Yes, both fragments are equally relevant	5
No, the fragment I chose is more relevant	4
No, the fragment suggested by the tool is more relevant	1
Total	10

Influence of Explanation on Fragment Selection (Q13)

When participants answered Q13, they had already seen the explanation supporting the recommended fragment. Q13 therefore assesses whether this additional information led to a change in participants' perception regarding the importance of the fragment suggested by the tool.

Table 21 presents the results for responses that initially disagreed with the recommended fragment (Q6 = No). Out of 10 responses, 3 ultimately accepted the recommen-

ded fragment in Q13, 4 responses maintained their original choice (Q13 = No), and 1 response was left blank. These findings suggest that explanatory information can positively influence participants' reconsideration of the recommended fragment, particularly when the fragment is already perceived as equally relevant to their own selection.

Table 21 – Change of Opinion from Q7 to Q13

Response in Q13	Count
Yes	3
No	4
No response	3
Total Responses	10

8.2.4.3 Importance of the Features Ranking

Agreement with the Ranking (Q16)

Participants were asked whether they agreed with the ranking presented in the section *Explanation based on Features (Ranking)*. Figure 24 shows the result: out of 26 responses, 16 (61.5%) fully agreed with the ranking, indicating strong alignment with the presented importance of the top five metrics. Additionally, 10 responses (38.5%) partially agreed with the ranking, while no responses disagreed or reported uncertainty. These results suggest that the ranking was generally perceived as consistent with the participants' evaluation of metric importance.

Perceived Helpfulness and Agreement with the Ranking

Table 22 shows a detailed examination of participant-level responses reveals that the same participant (P01) evaluated four different methods. Although this participant initially indicated that the metrics did not help for three of the four methods (Q3 = No), they ultimately fully agreed with the ranking presented in Q16 in all cases. This suggests that even when the perceived helpfulness of the metrics was low, the participant was able to align with the feature-based ranking, highlighting the potential influence of the explainability elements on decision-making at the individual level.

8.2.4.4 Assessment of Participant Confidence

The confidence of participants in their refactoring decisions was assessed both before (Q10) and after (Q19) considering the explainability elements. Descriptive statistics

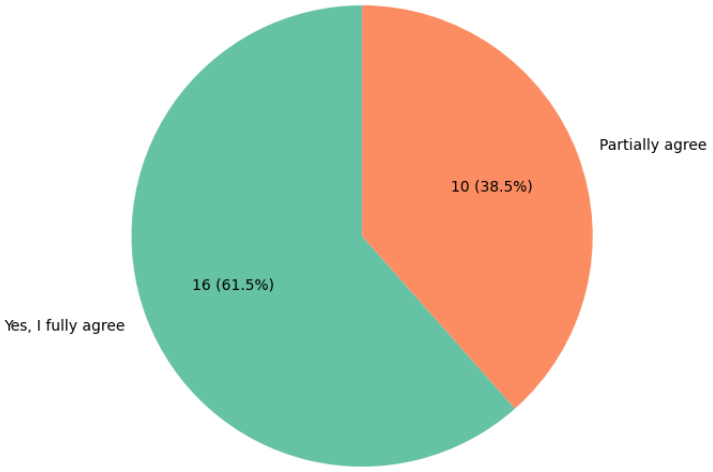


Figure 24 – Importance of Feature Ranking

Table 22 – Individual-Level Responses agreement with Ranking (Q16)

Spreadsheet	Participant	Q3 (Metrics Helpful)	Q16 (Agreement with Ranking)
G02-CB-M02	P01	Not answered	Yes, I fully agree
G02-CA-M03	P01	No	Yes, I fully agree
G02-CA-M04	P01	No	Yes, I fully agree
G02-CB-M01	P01	No	Yes, I fully agree

indicated an increase in the central tendency of confidence scores: the mean rose from 3.88 in Q10 to 4.15 in Q19, while the median remained at 4.0 for both questions. Additionally, the standard deviation decreased from 0.77 to 0.61, suggesting that participants' responses became more consistent after exposure to the explainability elements.

Although the analysis was conducted through the Paired Samples T-Test module in JASP (JASP Team, 2025), the software automatically selected the non-parametric *Wilcoxon signed-rank test* due to the ordinal nature of the Likert-scale data and the relatively small sample size ($n = 26$). The Wilcoxon test yielded a statistic of $W = 0.0$ and a p-value of 0.011 (two-sided, continuity correction applied), indicating that **the increase in confidence was statistically significant**. This result demonstrates that the explainability elements had a measurable positive impact on participants' perceived

reliability of their refactoring decisions.

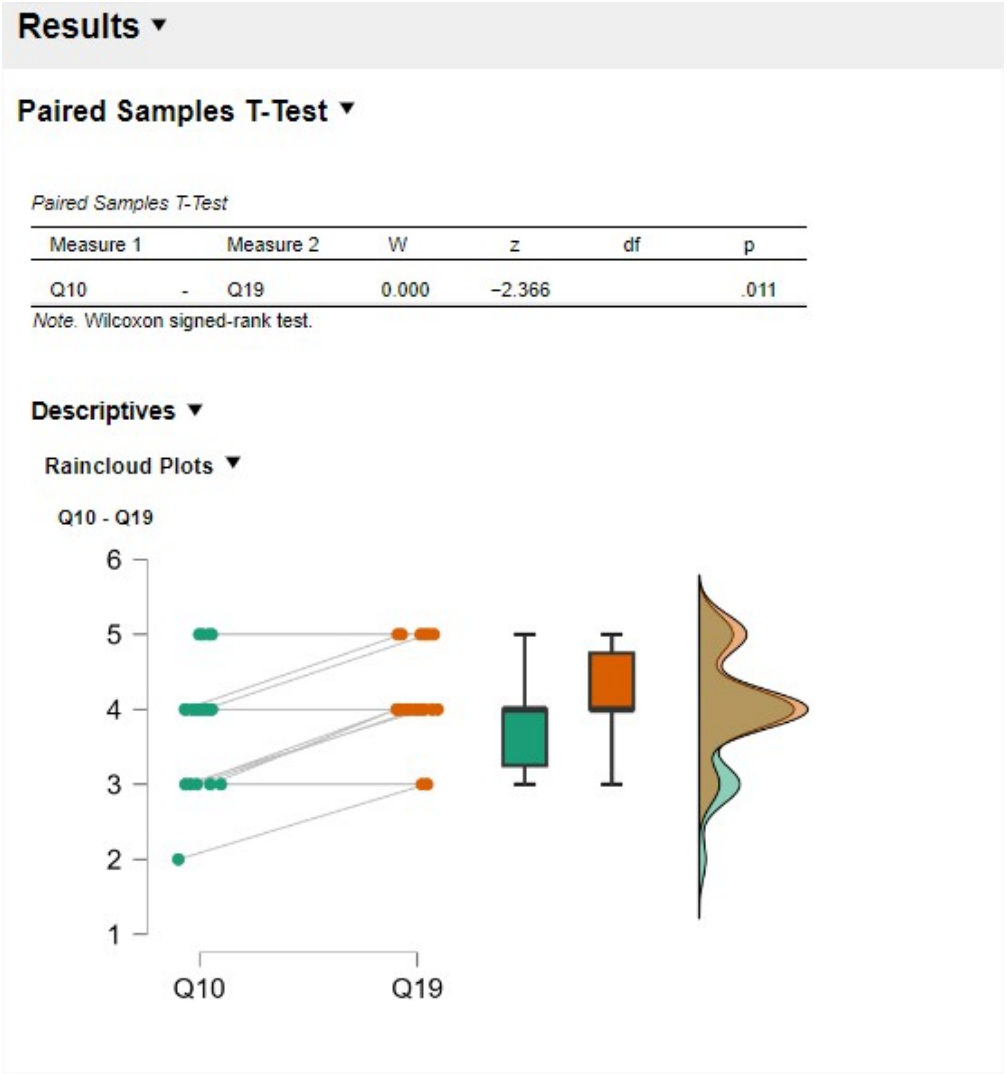


Figure 25 – JASP result - Confidence Q10 - Q19

Figure 25 presents the results generated by JASP. The Raincloud plot illustrates the distribution and paired differences between the confidence scores before (Q10) and after (Q19) the inclusion of the explainability elements. The visualization integrates raw data points, boxplots, and kernel density estimates, offering a comprehensive depiction of central tendency, variability, and distributional shape. Individual paired observations are connected by lines, highlighting the direction and magnitude of change for each

participant. The boxplots indicate that the median remained at 4 in both measurements, while the interquartile range suggests a modest increase in variability for Q19. The density plots on the right show a shift toward higher confidence values after the intervention, reinforcing the observed upward trend. Overall, the Raincloud plot effectively illustrates both the paired nature of the data and the distributional changes resulting from the intervention.

8.2.4.5 Expert (G1) vs. Non-Expert (G2) Response Analysis

In this study, participants were divided into two groups based on programming expertise: G01 (experts) and G02 (non-experts). Each participant evaluated four software methods. To assess whether the distribution of responses differed between the groups, statistical tests appropriate for each data type were applied. Binary questions (0/1) were analyzed using Fisher's exact test, suitable for small and unequal sample sizes. Questions with multiple categorical levels were examined with the Chi-squared test of independence, complemented by Cramer's V to quantify the strength of association. Likert-scale questions were analyzed with the non-parametric Mann-Whitney U test, along with the effect size (r).

Table 23 – Statistical analysis between expert (G01) and non-expert (G02) participants.

Question	Type	p-value	effect size
Q1	Binary	1.000	NA
Q3	Binary	0.250	NA
Q6	Binary	0.678	NA
Q11	Binary	0.300	NA
Q13	Binary	0.524	NA
Q15	Binary	0.083	NA
Q17	Binary	0.385	NA
Q7	Categorical	0.031	0.769
Q8	Categorical	NaN	NaN
Q16	Categorical	0.009	0.513
Q10	Likert	0.4334	0.175
Q19	Likert	0.026	0.463

Table 23 summarizes the results of the tests performed using the JASP tool (JASP Team, 2025). Among the categorical questions, Q7 and Q16 showed statistically

significant differences between experts and non-experts, with Cramer's V values of 0.77 and 0.51, respectively, indicating moderate to strong associations. Question Q8 did not provide valid results (NaN) and is thus not considered.

For Likert-scale questions, Q19 was significant ($p = 0.026$, $r = 0.463$), suggesting a moderate effect of expertise, whereas Q10 showed no significant difference ($p = 0.4334$, $r = 0.175$). The remaining binary questions (Q1, Q3, Q6, Q11, Q13, Q15, Q17) had p-values consistently above 0.05, implying no systematic influence of programming expertise on these items.

Overall, the results reveal observable differences between experts and non-experts, particularly in Q7 (perceived relevance of the code fragment) and Q16 (agreement with the feature-based ranking). These findings suggest that programming expertise may shape participants' evaluations; however, additional data and further testing are necessary to validate these differences and confirm their robustness.

8.2.4.6 Analysis of Participant Responses by Cluster (CA vs CB)

Table 24 summarizes the comparison of participant responses between clusters CA (high-metric methods) and CB (low-metric methods). For all binary questions (Q1, Q3, Q6, Q11, Q13, Q15, Q17), the p-values are well above the conventional significance level of 0.05, indicating no statistically significant differences between clusters. Similarly, for categorical questions (Q7, Q8, Q16) and Likert-scale questions (Q10, Q19), the p-values remain above 0.05, and the corresponding effect sizes (Cramer's V for categorical variables and r for Likert) are very low.

These results suggest that the cluster classification of methods, based on metric levels, does not meaningfully influence participants' evaluations. In other words, participants' judgments appear consistent regardless of whether a method belongs to the CA or CB cluster. This outcome implies that, for this set of methods and participants, the metrics-based clustering did not create perceptible differences in perceived relevance or agreement with evaluated aspects.

Table 24 – Comparison between Clusters (CA vs CB)

Question	Type	p-value	Effect Size
Q1_num	Binary	1.000	NA
Q3_num	Binary	0.593	NA
Q6_num	Binary	0.688	NA
Q11_num	Binary	1.000	NA
Q13_num	Binary	1.000	NA
Q15_num	Binary	1.000	NA
Q17_num	Binary	1.000	NA
Q7_num	Categorical	0.543	0.378
Q8_num	Categorical	NaN	NaN
Q16_num	Categorical	0.263	0.220
Q10	Likert	0.910	0.030
Q19	Likert	0.976	-0.012

8.3 Hypotheses Evaluation

Based on the collected data and statistical analyses, the three hypotheses can be evaluated as follows:

- **H_1:** *Providing a complete recommendation supported of our tool increases the acceptance rate of refactoring suggestions compared to the phase without tool support.*
  The analysis of Q1, Q8, and Q11 showed that one participant revised their decision from "No" to "Yes" after being exposed to the explanation, while another participant maintained a negative judgment. Although this result suggests that explanatory support may positively influence acceptance, the number of relevant cases (n = 2) is too limited to draw generalizable conclusions. Therefore, H_1 is **partially supported in an exploratory sense**, highlighting the need for larger-scale studies to confirm the effect.

- **H_2:** *Providing a complete recommendation improves users' perception of confidence in the recommendation.*
  Confidence levels, measured before (Q10) and after (Q19) exposure to explainability, increased from a mean of 3.88 to 4.15. A Wilcoxon signed-rank test confirmed the statistical significance of this improvement ($p = 0.011$), and the reduced stan-

dard deviation indicates more consistent participant evaluations. These results provide **empirical support for H_2**, demonstrating that explainability elements significantly enhance perceived reliability and confidence.

In summary, the evaluation provides **robust evidence supporting H_2**, whereas H_1 receives more limited forms of support. Specifically, **partial support** indicates that the findings are consistent with the hypothesis but not strong or consistent enough to be considered conclusive, often due to modest effects or sample size limitations. In contrast, **exploratory support** reflects preliminary evidence that should be interpreted with caution and mainly regarded as a basis for generating new hypotheses or guiding future research. Overall, the results highlight the complete recommendation enhancing user confidence and acceptance, while also underscoring the need for studies with larger participant samples to consolidate and validate these findings statistically.

8.3.1 Threats to Validity

Construct Validity: Our survey aimed to capture participants' perceptions, confidence, and acceptance of recommended refactorings. While Likert, categorical, and binary questions were used, responses may not fully reflect true decision-making.

Internal Validity: With only seven participants, statistical power is limited. The two phases design can introduce learning and carryover effects, where familiarity with the task or decisions from the first phase influences the second.

External Validity: The dataset included only 8 methods, constrained by LOC and FME-recognizable structures, limiting generalization to methods of different sizes or complexities.

Conclusion Validity: The small number of participants and the conditional nature of some survey questions (Q8, Q11, Q13) reduced the number of valid responses, limiting formal statistical analysis. Observed trends should thus be interpreted as exploratory.

Mitigation Strategies: To reduce these threats, methods from both C-low and C-high clusters were randomly assigned to forms and participants, and instructions were standardized across phases. Furthermore, different methods were used in each phase to

minimize carryover, ensuring that prior exposure to one method did not directly influence decisions on the next.

8.4 Summary of Findings

The evaluation of the Refactoring Recommendation Tool yielded several important findings:

- Providing complete recommendations showed limited but positive evidence of increasing acceptance (H_1), although the small number of relevant cases restricts generalization.
- Providing complete recommendations enhanced participants' confidence in their refactoring decisions, with statistical evidence supporting H_2.
- The tool's recommended fragments were perceived as aligned with participants' own choices in the majority of cases (57.7%).
- Benefits of applying the Extract Method, participants widely agree with the benefits (96% agreement), highlighting words as improvements, readability, maintainability, modularity, and code organization.
- Experts and non-experts differed in their perception of fragment relevance (Q7) and in their agreement with the feature ranking (Q16), suggesting that expertise shapes evaluation.
- No significant differences were found between clusters of high- and low-metric methods, indicating that tool evaluations were consistent across different method profiles.

Overall, these results suggest that the tool effectively supports user confidence and provides generally relevant recommendations, while highlighting areas that require further validation with larger participant samples.

Chapter 9
CONCLUSION

9.1 Initial Considerations

This chapter concludes the thesis by revisiting its main motivation and objectives, and by highlighting the contributions achieved. The central aim of this research was to address the problem of *incomplete recommendations for Extract Method* refactoring, identified in the literature and characterized by providing the user with partial elements of a recommendation.

The work was guided by three main motivations: (i) extending the scope of recommendations beyond traditional code smells to reflect real refactoring practices, (ii) improving the formulation of recommendations to integrate the full set of W3B criteria (Which, Where, Why, Benefits), and (iii) enhancing the interpretability and trustworthiness of recommendations through Explainable AI techniques.

Accordingly, the chapter is structured as follows: the Contributions section 9.2 summarizes the key scientific and technical achievements, the Limitations section 9.3 discusses the boundaries and constraints of the proposed approach, the Future Work section 9.4 presents directions for continued research, and the Final Remarks 9.5 section synthesizes the significance and impact of the work.

9.2 Contributions

We summarize our main contributions as follows:

1. **Systematic Review of Refactoring Recommendations**
   A systematic review was conducted in Chapter 3, on the use of machine learning for refactoring recommendations, analyzing 22 approaches. It revealed a strong focus on *Extract Method* and *Move Method* refactorings, while a large majority of Fowler's 67 refactorings remain largely unexplored. The review showed that supervised learning algorithms, particularly Random Forest, are predominant, and Refactoring-based (RB) datasets are commonly used to capture real-world refactoring motivations. Critically, the chapter identified that most approaches are semi-automated and often provide incomplete recommendations.

2. **W3B Criteria for Recommendation Completeness**
   This thesis proposes the *W3B criteria (Which, Where, Why, and Benefits)* to assess the completeness of refactoring recommendations. These criteria define each element: Which refactoring to apply, Where in the code it should be applied, Why it is recommended, and the Benefits it brings. This criterion is also the result of the systematic review conducted. Thus, applying W3B to the systematic review (Chapter 3) highlighted a significant deficiency: most existing approaches neglect the *Why* and *Benefits* criteria, with only a small percentage satisfying all four. This omission is critical, as it undermines developer confidence and limits the practical adoption of automated tools.

3. **Consensual Explainer**
   To address inconsistencies among Explainable AI (XAI) techniques, this thesis introduces the Consensual Explainer, which aggregates outputs from multiple explainers into a unified, more stable, and reliable explanation. The module employs a systematic process (Chapter 5), including filtering, sorting, and defining a *Priority Order of Explainers* (POExp) to resolve ranking conflicts. Empirical analysis demonstrated that this approach consistently achieves higher agreement levels (FA metric) compared to individual explainers or baseline techniques, particularly in scenarios with higher feature complexity.

4. **Refactoring Recommendation API for Extract Method**
   We implemented a Refactoring Recommendation API, aiming to generate *Complete Extract Method recommendations* based on real extractions. The methodo-

logical process includes four phases. Phase I focuses on building a specialized, balanced dataset of method-fragment pairs, using cosine distance for robust negative sample selection. Phase II fine-tunes a CodeBERT model for regression, achieving high predictive performance in identifying extract method candidates. Phase III integrates the Consensual Explainer, utilizing a Random Forest surrogate for interpretability. Phase IV assembles the tool, employing the Fragment Metric Extractor (FME), the trained models, and GPT-4 to generate natural language recommendations aligned with the W3B criteria.

These contributions collectively advance both the scientific understanding and practical application of refactoring recommendations.

9.3 Limitations of the approach

Despite the contributions, the proposed approach has some limitations:

- **Scope restricted to Extract Method:** The tool is exclusively dedicated to recommending Extract Method refactorings, which constrains its applicability to other types of refactoring.
- **Dependency on external tools and APIs:** The approach requires an active internet connection to (i) retrieve source code from public GitHub repositories and (ii) invoke the GPT-4 API to generate natural language recommendations. These dependencies impose restrictions on usage, as only public repositories are supported.
- **Fragment Metric Extractor (FME) limitations:** The JAR is unable to process methods that involve the creation of anonymous classes or subclasses. Additionally, when overloaded methods with the same name are present, none of them are processed, since the extractor relies solely on the method name as input.
- **Quality and representativeness of training data:** Although a specialized dataset was constructed, the system's effectiveness is inherently tied to the quality of the training data and the characteristics of the analyzed projects. As the datasets were derived from open-source Java projects, generalization to other programming languages, industry contexts, or domains may be limited.

These limitations do not undermine the contributions of this thesis but highlight opportunities for improvement and future work.

9.4 Future Work

Building upon the results and limitations, several promising directions for future research and development emerge:

- The Fragment Metric Extractor (FME) presents several avenues for enhancement. Future work may include extending its ability to handle anonymous classes and overloaded methods, as well as exploring alternative strategies for fragment identification. In addition, going beyond the conventional set of twenty metric by incorporating process-oriented measures and historical data, could provide a more complete picture of coding practices and their context.
- The W3B criteria could be refined by introducing quantitative measures that evaluate the completeness of a recommendation. This would shift the framework from being mainly qualitative towards a more precise basis for evaluation and comparison across different approaches.
- Integrating the tool into IDEs is a key step for practical adoption. This will require solving methodological and technical challenges, such as optimizing model loading to reduce response time. Although the current prototype works as a web interface, providing it as an IDE plugin will be crucial to ensure smooth integration into developers' workflows and to increase its applicability in real-world settings.
- Adding robust user feedback mechanisms is another promising direction. Such mechanisms would allow the tool to adapt to developers' preferences, making recommendations more relevant and improving their acceptance in practice. An adaptive feedback loop would also help build trust and encourage long-term use.
- Large-scale empirical evaluations are needed to strengthen external validity and generalizability. The current controlled experiment, was limited in terms of participants (seven practitioners) and scope (eight methods). Expanding the evaluation

to include more participants, diverse project types, and industrial contexts will provide stronger evidence of the tool's effectiveness and practical impact.

9.5 Final Remarks

This work represents an important step toward overcoming the limitations of refactoring recommendation approaches/tools. By proposing the **W3B criteria** and developing a modular tool, we demonstrated the feasibility of generating **complete and understandable Extract Method recommendations** grounded in real-world practices. Beyond technical robustness, the approach emphasizes developer trust, which is crucial for practical adoption.

The integration of the CodeBERT model with Explainable AI techniques shows the value of combining semantic understanding with interpretability in software engineering tools. While the evaluation yielded promising results, it also revealed opportunities for improvement, such as incorporating additional models and refining the Fragment Metric Extractor, which open avenues for future research.

Overall, this thesis helps bridge the gap between theoretical advancements and practical applications in software engineering. It highlights the potential of AI to address challenges in code maintainability and developer productivity by trusting and accepting recommendations, providing a foundation for more comprehensive and developer-centric solutions.

REFERÊNCIAS

ACHIAM, J.; ADLER, S.; AGARWAL, S.; AHMAD, L.; AKKAYA, I.; ALEMAN, F. L.; ALMEIDA, D.; ALTENSCHMIDT, J.; ALTMAN, S.; ANADKAT, S. et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. Citado na página 129.

ADVANSE-LAB. *Source code of the Consensus Module*. 2024. <https://github.com/Advanse-Lab/consensus-explanability> [Accessed: 11/06/2025]. Citado na página 100.

ALENEZI, M.; AKOUR, M.; QASEM, O. A. Harnessing deep learning algorithms to predict software refactoring. *Telkomnika*, Ahmad Dahlan University, v. 18, n. 6, p. 2977–2982, 2020. Citado 2 vezes nas páginas 24 e 52.

ALIZADEH, V.; FEHRI, H.; KESSENTINI, M. Less is more: From multi-objective to mono-objective refactoring via developer's knowledge extraction. In: IEEE. *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.], 2019. p. 181–192. Citado 4 vezes nas páginas 21, 22, 51 e 75.

ALIZADEH, V.; KESSENTINI, M. Reducing interactive refactoring effort via clustering-based multi-objective search. In: IEEE. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2018. p. 464–474. Citado 2 vezes nas páginas 21 e 52.

ALIZADEH, V.; OUALI, M. A.; KESSENTINI, M.; CHATER, M. Refbot: Intelligent software refactoring bot. In: IEEE. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2019. p. 823–834. Citado 2 vezes nas páginas 30 e 67.

ALOMAR, E. A.; IVANOV, A.; KURBATOVA, Z.; GOLUBEV, Y.; MKAOUER, M. W.; OUNI, A.; BRYKSIN, T.; NGUYEN, L.; KINI, A.; THAKUR, A. Just-in-time code duplicates extraction. *Information and Software Technology*, Elsevier, v. 158, 2023. Citado 7 vezes nas páginas 21, 23, 51, 75, 84, 86 e 98.

ANGELOV, P. P.; SOARES, E. A.; JIANG, R.; ARNOLD, N. I.; ATKINSON, P. M. Explainable artificial intelligence: an analytical review. *Wiley Interdisciplinary Reviews:*

Data Mining and Knowledge Discovery, Wiley Online Library, v. 11, n. 5, p. e1424, 2021. Citado 2 vezes nas páginas 10 e 36.

ANGULO, G. *Investigating the Employment of ML for Extract Method Recommendation*. 2025. Repository. Repository: <https://tinyurl.com/46z9asmk>
FME: <https://github.com/guiseAA/Fragment-Metric-Extractor_FME>
Dataset: <https://tinyurl.com/mw5ap3xa>
Notebooks: <https://github.com/guiseAA/Notebooks_PhDproject/tree/main>
Fine-Tuning: <https://github.com/guiseAA/Fine-Tuning_Cross-Validation/tree/main>. Citado 3 vezes nas páginas 117, 121 e 135.

ANGULO, G. *Refactoring Recommendation Tool*. 2025. <https://sites.google.com/view/recommendertool/inicio>. Accessed: 2025-08-18. Citado 4 vezes nas páginas 152, 155, 157 e 159.

ANICHE; MAZIERO; RAFAEL; VINICIUS. *ML 4 Refactoring: dataset*. Zenodo, 2019. Disponível em: <https://doi.org/10.5281/zenodo.3547639>. Citado na página 108.

ANICHE, M. *Java code metrics calculator (CK)*. [S.l.], 2015. Available in https://github.com/mauricioaniche/ck/. Citado na página 126.

ANICHE, M.; MAZIERO, E.; DURELLI, R.; DURELLI, V. H. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, IEEE, v. 48, n. 4, p. 1432–1450, 2020. Citado 4 vezes nas páginas 52, 61, 97 e 111.

ARMIJO, G.; SANTIBAñEZ, D.; DURELLI, R.; CAMARGO, V. On the employment of machine learning for recommending refactorings: A systematic literature review. In: *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*. Porto Alegre, RS, Brasil: SBC, 2024. ISSN 0000-0000. Citado na página 86.

BANEGAS-LUNA, A. J.; MARTINEZ-CORTES, C.; PEREZ-SANCHEZ, H. Fighting the disagreement in explainable machine learning with consensus. *arXiv preprint arXiv:2307.01288*, 2023. Citado na página 86.

BAVOTA, G.; LUCIA, A. D.; MARCUS, A.; OLIVETO, R. Recommending refactoring operations in large software systems. In: *Recommendation Systems in Software Engineering*. [S.l.]: Springer, 2014. p. 387–419. Citado na página 30.

BOMMER, P. L.; KRETSCHMER, M.; HEDSTRöM, A.; BAREEVA, D.; HöHNE, M. M.-C. Finding the right xai method—a guide for the evaluation and ranking of explainable ai methods in climate science. *Artificial Intelligence for the Earth Systems*, American Meteorological Society, Boston MA, USA, v. 3, n. 3, 2024. Citado 2 vezes nas páginas 24 e 86.

CELEBI, M. E.; AYDIN, K. *Unsupervised learning algorithms*. [S.l.]: Springer, 2016. Citado na página 32.

CHARALAMPIDOU, S.; AMPATZOGLOU, A.; CHATZIGEORGIOU, A.; GKORTZIS, A.; AVGERIOU, P. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering*, IEEE, v. 43, n. 10, p. 954–974, 2016. Citado 2 vezes nas páginas 63 e 66.

CUI, D.; WANG, Q.; WANG, S.; CHI, J.; LI, J.; WANG, L.; LI, Q. Rems: Recommending extract method refactoring opportunities via multi-view representation of code property graph. In: IEEE. *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. [S.l.], 2023. Citado 9 vezes nas páginas 21, 23, 51, 75, 84, 86, 98, 122 e 123.

CUI, D.; WANG, S.; LUO, Y.; LI, X.; DAI, J.; WANG, L.; LI, Q. Rmove: Recommending move method refactoring opportunities using structural and semantic representations of code. In: IEEE. *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2022. p. 281–292. Citado 3 vezes nas páginas 52, 122 e 123.

CUNHA, W. S.; ARMIJO, G. A.; CAMARGO, V. V. de. Investigating non-usually employed features in the identification of architectural smells: A machine learning-based approach. In: . New York, NY, USA: Association for Computing Machinery, 2020. (SBCARS '20). ISBN 9781450387545. Citado na página 86.

DAS, A.; RAD, P. Opportunities and challenges in explainable artificial intelligence (xai): A survey. *arXiv preprint arXiv:2006.11371*, 2020. Citado na página 76.

DEVLIN, J.; CHANG, M.-W.; LEE, K.; TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. [S.l.: s.n.], 2019. p. 4171–4186. Citado na página 33.

DIEBER, J.; KIRRANE, S. Why model why? assessing the strengths and limitations of lime. *arXiv preprint arXiv:2012.00093*, 2020. Citado 2 vezes nas páginas 37 e 123.

FENG, Z.; GUO, D.; TANG, D.; DUAN, N.; FENG, X.; GONG, M.; SHOU, L.; QIN, B.; LIU, T.; JIANG, D. et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. Citado 4 vezes nas páginas 22, 33, 34 e 117.

FICO. *Explainable Machine Learning Challenge*. 2018. Website of FICO Community. Disponível em: <https://community.fico.com/s/explainable-machine-learning-challenge>. Citado na página 35.

FOKAEFS, M.; TSANTALIS, N.; CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of feature envy bad smells. In: IEEE. *2007 ieee international conference on software maintenance*. [S.l.], 2007. p. 519–520. Citado 3 vezes nas páginas 21, 62 e 66.

FOLWER, M. Refactoring: Improving the design of existing programs.(1999). *Google Scholar Google Scholar Digital Library Digital Library*, 1999. Citado na página 29.

FONTANA, F. A.; MÄNTYLÄ, M. V.; ZANONI, M.; MARINO, A. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, Springer, v. 21, p. 1143–1191, 2016. Citado na página 61.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 2018. Citado 2 vezes nas páginas 20 e 30.

FOWLER, M.; BECK, K. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019. (A Martin Fowler signature book). ISBN 9780134757599. Disponível em: <https://books.google.com.br/books?id=o69NtAEACAAJ>. Citado na página 53.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. Refactoring: Improving the design of existing code. *Google Scholar Google Scholar Digital Library Digital Library*, 1999. Citado 2 vezes nas páginas 20 e 29.

GOHEL, P.; SINGH, P.; MOHANTY, M. Explainable ai: current status and future directions. *arXiv preprint arXiv:2107.07045*, 2021. Citado na página 35.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. Citado na página 22.

GUISELLA, A.; DANIEL, S. M. S.; VALTER, V. d. C.; RAFAEL, D. *On the Employment of Machine Learning for Recommending Refactorings: A Systematic Literature Review*. Zenodo, 2024. Disponível em: <https://doi.org/10.5281/zenodo.11406434>. Citado na página 43.

GUNNING, D. Explainable artificial intelligence (xai). *Defense advanced research projects agency (DARPA), nd Web*, v. 2, n. 2, p. 1, 2017. Citado na página 35.

GUNNING, D.; STEFIK, M.; CHOI, J.; MILLER, T.; STUMPF, S.; YANG, G.-Z. Xai—explainable artificial intelligence. *Science Robotics*, American Association for the Advancement of Science, v. 4, n. 37, p. eaay7120, 2019. Citado 4 vezes nas páginas 24, 35, 86 e 121.

HASSIJA, V.; CHAMOLA, V.; MAHAPATRA, A.; SINGAL, A.; GOEL, D.; HUANG, K.; SCARDAPANE, S.; SPINELLI, I.; MAHMUD, M.; HUSSAIN, A. Interpreting black-box models: a review on explainable artificial intelligence. *Cognitive Computation*, Springer, v. 16, n. 1, 2024. Citado na página 24.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. et al. *The elements of statistical learning*. [S.l.]: Springer series in statistics New-York, 2009. Citado na página 32.

HENRIQUE, J.; DÓSEA, M.; SANT'ANNA, C. Minerando motivações para aplicação de extract method: Um estudo preliminar. In: SBC. *Anais do IX Workshop de Visualização, Evolução e Manutenção de Software*. [S.l.], 2021. p. 21–25. Citado 3 vezes nas páginas 30, 107 e 131.

HENRIQUE, J. d. S. et al. Motivações para aplicação da refatoração extract method: um estudo baseado em mensagens de commit. Universidade Federal da Bahia, 2023. Citado na página 21.

IMAZATO, A.; HIGO, Y.; HOTTA, K.; KUSUMOTO, S. Finding extract method refactoring opportunities by analyzing development history. In: IEEE. *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. [S.l.], 2017. v. 1, p. 190–195. Citado 3 vezes nas páginas 21, 53 e 98.

ISLAM, M. R.; AHMED, M. U.; BARUA, S.; BEGUM, S. A systematic review of explainable artificial intelligence in terms of different application domains and tasks. *Applied Sciences*, MDPI, v. 12, n. 3, p. 1353, 2022. Citado na página 35.

JASP Team. *JASP (Version 0.95.0)[Computer software]*. 2025. Disponível em: <https://jasp-stats.org/>. Citado 2 vezes nas páginas 163 e 165.

KAUR, M.; RATTAN, D. A systematic literature review on the use of machine learning in code clone research. *Computer Science Review*, Elsevier, v. 47, 2023. Citado na página 86.

KEELE, S. et al. *Guidelines for performing systematic literature reviews in software engineering*. [S.l.], 2007. Citado 2 vezes nas páginas 26 e 39.

KIM, Y.; KIM, Y. Explainable heat-related mortality with random forest and shapley additive explanations (shap) models. *Sustainable Cities and Society*, Elsevier, v. 79, p. 103677, 2022. Citado na página 122.

KOTSIANTIS, S. B.; ZAHARAKIS, I.; PINTELAS, P. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, v. 160, p. 3–24, 2007. Citado na página 31.

KRISHNA, S.; HAN, T.; GU, A.; WU, S.; JABBARI, S.; LAKKARAJU, H. The disagreement problem in explainable machine learning: A practitioner's perspective. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. Citado 3 vezes nas páginas 86, 89 e 97.

KRISHNA, S.; MA, J.; SLACK, D.; GHANDEHARIOUN, A.; SINGH, S.; LAKKARAJU, H. Post hoc explanations of language models can improve language models. In: OH, A.; NAUMANN, T.; GLOBERSON, A.; SAENKO, K.; HARDT, M.; LEVINE, S. (Ed.). *Advances in Neural Information Processing Systems*. [S.l.]: Curran Associates, Inc., 2023. v. 36. Citado na página 97.

KUMAR, L.; SATAPATHY, S. M.; KRISHNA, A. Application of smote and lssvm with various kernels for predicting refactoring at method level. In: SPRINGER. *International Conference on Neural Information Processing*. [S.l.], 2018. p. 150–161. Citado na página 53.

KUMAR, L.; SATAPATHY, S. M.; SUREKA, A. *Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques*. [S.l.]: Academic Press, 2015. Citado 2 vezes nas páginas 24 e 52.

KUMAR, L.; SUREKA, A. Application of lssvm and smote on seven open source projects for predicting refactoring at class level. In: IEEE. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.], 2017. p. 90–99. Citado 3 vezes nas páginas 21, 24 e 53.

KURBATOVA, Z.; VESELOV, I.; GOLUBEV, Y.; BRYKSIN, T. Recommendation of move method refactoring using path-based representation of code. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. [S.l.: s.n.], 2020. p. 315–322. Citado 3 vezes nas páginas 21, 23 e 52.

LABERGE, G.; PEQUIGNOT, Y. B.; MARCHAND, M.; KHOMH, F. Tackling the xai disagreement problem with regional explanations. In: PMLR. *International Conference on Artificial Intelligence and Statistics*. [S.l.], 2024. Citado na página 86.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, IEEE, v. 68, n. 9, p. 1060–1076, 1980. Citado na página 20.

LEIJ, D. van der; BINDA, J.; DALEN, R. van; VALLEN, P.; LUO, Y.; ANICHE, M. Data-driven extract method recommendations: a study at ing. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2021. p. 1337–1347. Citado 3 vezes nas páginas 52, 122 e 123.

LIU, H.; XU, Z.; ZOU, Y. Deep learning based feature envy detection. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. [S.l.: s.n.], 2018. p. 385–396. Citado 3 vezes nas páginas 21, 23 e 53.

LONGO, L.; GOEBEL, R.; LECUE, F.; KIESEBERG, P.; HOLZINGER, A. Explainable artificial intelligence: Concepts, applications, research challenges and visions. In: SPRINGER. *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*. [S.l.], 2020. p. 1–16. Citado 2 vezes nas páginas 35 e 121.

LUNDBERG, S.; LEE, S.-I. *A Unified Approach to Interpreting Model Predictions*. 2017. Disponível em: <https://arxiv.org/abs/1705.07874>. Citado 2 vezes nas páginas 88 e 123.

LUNDBERG, S. M.; LEE, S.-I. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, v. 30, 2017. Citado 3 vezes nas páginas 24, 38 e 86.

MA, W.; YU, Y.; RUAN, X.; CAI, B. Pre-trained model based feature envy detection. In: IEEE. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2023. p. 430–440. Citado na página 110.

MENS, T.; TOURWÉ, T. A survey of software refactoring. *IEEE Transactions on software engineering*, IEEE, v. 30, n. 2, p. 126–139, 2004. Citado 2 vezes nas páginas 20 e 30.

MICHALSKI, R.; CARBONELL, J.; MITCHELL, T. *Machine Learning: Artificial Intelligence Approach*. 1985. Citado na página 31.

MURPHY, K. P. *Machine learning: a probabilistic perspective*. [S.l.]: MIT press, 2012. Citado na página 32.

NYAMAWE, A. S. Mining commit messages to enhance software refactorings recommendation: A machine learning approach. *Machine Learning with Applications*, Elsevier, p. 100316, 2022. Citado 5 vezes nas páginas 23, 24, 51, 75 e 84.

NYAMAWE, A. S.; LIU, H.; NIU, N.; UMER, Q.; NIU, Z. Automated recommendation of software refactorings based on feature requests. In: IEEE. *2019 IEEE 27th International Requirements Engineering Conference (RE)*. [S.l.], 2019. p. 187–198. Citado na página 51.

NYAMAWE, A. S.; LIU, H.; NIU, N.; UMER, Q.; NIU, Z. Feature requests-based recommendation of software refactorings. *Empirical Software Engineering*, Springer, v. 25, n. 5, p. 4315–4347, 2020. Citado 3 vezes nas páginas 21, 24 e 51.

NYIRONGO, B.; JIANG, Y.; JIANG, H.; LIU, H. A survey of deep learning based software refactoring. *arXiv preprint arXiv:2404.19226*, 2024. Citado na página 22.

OPDYKE, W. F. Refactoring object-oriented frameworks. Citeseer, 1992. Citado na página 29.

PALIT, I.; SHETTY, G.; ARIF, H.; SHARMA, T. Automatic refactoring candidate identification leveraging effective code representation. In: IEEE. *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2023. p. 369–374. Citado 5 vezes nas páginas 51, 75, 98, 122 e 123.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. D. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In: *Proceedings of the 40th International Conference on Software Engineering*. [S.l.: s.n.], 2018. p. 482–482. Citado na página 61.

PANIGRAHI, R.; KUANAR, S. K.; MISRA, S.; KUMAR, L. Class-level refactoring prediction by ensemble learning with various feature selection techniques. *Applied Sciences*, Multidisciplinary Digital Publishing Institute, v. 12, n. 23, p. 12217, 2022. Citado 2 vezes nas páginas 24 e 52.

PANIGRAHI, R.; KUMAR, L. et al. Application of naïve bayes classifiers for refactoring prediction at the method level. In: IEEE. *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*. [S.l.], 2020. p. 1–6. Citado 3 vezes nas páginas 21, 23 e 52.

PANTIUCHINA, J.; LIN, B.; ZAMPETTI, F.; PENTA, M. D.; LANZA, M.; BAVOTA, G. Why do developers reject refactorings in open-source projects? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, v. 31, n. 2, p. 1–23, 2021. Citado 2 vezes nas páginas 73 e 86.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, É. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, v. 12, p. 2825–2830, 2011. Citado na página 116.

PETSIUK, V.; DAS, A.; SAENKO, K. Rise: Randomized input sampling for explanation of black-box models. *ArXiv*, abs/1806.07421, 2018. Citado na página 88.

PHILLIPS, P. J.; HAHN, C. A.; FONTANA, P. C.; BRONIATOWSKI, D. A.; PRZYBOCKI, M. A. Four principles of explainable artificial intelligence. *Gaithersburg, Maryland*, 2020. Citado na página 35.

PIRIE, C.; WIRATUNGA, N.; WIJEKOON, A.; MORENO-GARCIA, C. F. Agree: a feature attribution aggregation framework to address explainer disagreements with alignment metrics. In: *CEUR Workshop Proceedings*. [S.l.: s.n.], 2023. Citado 2 vezes nas páginas 86 e 122.

REBAI, S.; ALIZADEH, V.; KESSENTINI, M.; FEHRI, H.; KAZMAN, R. Enabling decision and objective space exploration for interactive multi-objective refactoring. *IEEE Transactions on Software Engineering*, IEEE, 2020. Citado 3 vezes nas páginas 51, 75 e 84.

RIBEIRO, M.; SINGH, S.; GUESTRIN, C. Anchors: High-precision model-agnostic explanations. *Proceedings of the AAAI Conference on Artificial Intelligence*, v. 32, 04 2018. Citado 6 vezes nas páginas 24, 86, 88, 93, 100 e 123.

RIBEIRO, M. T.; SINGH, S.; GUESTRIN, C. "why should i trust you?"explaining the predictions of any classifier. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. [S.l.: s.n.], 2016. Citado 3 vezes nas páginas 24, 86 e 88.

RIBEIRO, M. T.; SINGH, S.; GUESTRIN, C. Anchors: High-precision model-agnostic explanations. In: *Proceedings of the AAAI conference on artificial intelligence*. [S.l.: s.n.], 2018. v. 32, n. 1. Citado na página 38.

ROY, S.; LABERGE, G.; ROY, B.; KHOMH, F.; NIKANJAM, A.; MONDAL, S. Why don't xai techniques agree? characterizing the disagreements between post-hoc explanations of defect predictions. In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2022. Citado 2 vezes nas páginas 24 e 86.

ROY, S.; LABERGE, G.; ROY, B.; KHOMH, F.; NIKANJAM, A.; MONDAL, S. Why don't xai techniques agree? characterizing the disagreements between post-hoc explanations of defect predictions. In: IEEE. *2022 IEEE international conference on software maintenance and evolution (ICSME)*. [S.l.], 2022. p. 444–448. Citado na página 122.

SAHEB-NASSAGH, R.; ASHTIANI, M.; MINAEI-BIDGOLI, B. A probabilistic-based approach for automatic identification and refactoring of software code smells. *Applied Soft Computing*, Elsevier, v. 130, p. 109658, 2022. Citado 3 vezes nas páginas 21, 23 e 52.

SALIH, A. M.; RAISI-ESTABRAGH, Z.; GALAZZO, I. B.; RADEVA, P.; PETERSEN, S. E.; LEKADIR, K.; MENEGAZ, G. A perspective on explainable artificial intelligence methods: Shap and lime. *Advanced Intelligent Systems*, Wiley Online Library, v. 7, n. 1, p. 2400304, 2025. Citado na página 122.

SALTON, G.; WONG, A.; YANG, C.-S. A vector space model for automatic indexing. *Communications of the ACM*, ACM New York, NY, USA, v. 18, n. 11, p. 613–620, 1975. Citado na página 115.

SCHWARZSCHILD, A.; CEMBALEST, M.; RAO, K.; HINES, K.; DICKERSON, J. Reckoning with the disagreement problem: Explanation consensus as a training objective. In: *Proceedings of the 2023 AAAI/ACM Conference on AI, Ethics, and Society*. New York, NY, USA: Association for Computing Machinery, 2023. (AIES '23). ISBN 9798400702310. Citado na página 86.

SHARMA, T.; SPINELLIS, D. A survey on software smells. *Journal of Systems and Software*, Elsevier, v. 138, p. 158–173, 2018. Citado na página 30.

SHENEAMER, A. M. An automatic advisor for refactoring software clones based on machine learning. *IEEE Access*, IEEE, v. 8, p. 124978–124988, 2020. Citado 3 vezes nas páginas 21, 23 e 52.

SIDHU, B. K.; SINGH, K.; SHARMA, N. A machine learning approach to software model refactoring. *International journal of computers and applications*, Taylor & Francis, v. 44, n. 2, p. 166–177, 2022. Citado 2 vezes nas páginas 24 e 52.

SILVA, C.; SANTANA, A.; FIGUEIREDO, E.; BIGONHA, M. A. Revisiting the bad smell and refactoring relationship: A systematic literature review. Citado na página 30.

SILVA, D.; TERRA, R.; VALENTE, M. T. Jextract: An eclipse plug-in for recommending automated extract method refactorings. *arXiv preprint arXiv:1506.06086*, 2015. Citado 2 vezes nas páginas 63 e 66.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of github contributors. In: *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. [S.l.: s.n.], 2016. p. 858–870. Citado na página 61.

SIMMONDS, J.; MENS, T. A comparison of software refactoring tools. *Programming Technology Lab*, 2002. Citado na página 67.

SUNDARARAJAN, M.; TALY, A.; YAN, Q. Axiomatic attribution for deep networks. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. [S.l.]: JMLR.org, 2017. (ICML'17). Citado na página 88.

SVYATKOVSKIY, A.; DENG, M.; FU, S.; SUNDARESAN, N. Intellicode compose: Code generation using transformer-based language models. In: *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2021. p. 1438–1449. Citado 2 vezes nas páginas 33 e 34.

SYMON, H. Neural networks: a comprehensive foundation. *Prentice-Hall*, 1999. Citado na página 33.

TERRA, R.; VALENTE, M. T.; MIRANDA, S.; SALES, V. Jmove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, Elsevier, v. 138, p. 19–36, 2018. Citado 3 vezes nas páginas 21, 63 e 66.

TSANTALIS, N.; KETKAR, A.; DIG, D. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, IEEE, v. 48, n. 3, 2020. Citado 2 vezes nas páginas 98 e 111.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, Ł.; POLOSUKHIN, I. Attention is all you need. In: *Advances in Neural Information Processing Systems (NeurIPS)*. [S.l.: s.n.], 2017. Citado na página 33.

VILONE, G.; LONGO2020EXPLAINABLE, L. Explainable artificial intelligence: a systematic review. *arXiv preprint arXiv:2006.00093*, 2020. Citado na página 122.

WOHLIN, C. Experimentation in software engineering: an introduction. *(No Title)*, 2000. Citado 2 vezes nas páginas 150 e 151.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. [S.l.: s.n.], 2014. p. 1–10. Citado na página 44.

XU, S.; GUO, C.; LIU, L.; XU, J. A log-linear probabilistic model for prioritizing extract method refactorings. In: IEEE. *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. [S.l.], 2017. p. 2503–2507. Citado 4 vezes nas páginas 21, 23, 53 e 98.

XU, S.; SIVARAMAN, A.; KHOO, S.-C.; XU, J. Gems: An extract method refactoring recommender. In: IEEE. *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. [S.l.], 2017. p. 24–34. Citado 2 vezes nas páginas 53 e 61.

YUE, R.; GAO, Z.; MENG, N.; XIONG, Y.; WANG, X.; MORGENTHALER, J. D. Automatic clone recommendation for refactoring based on the present and the past. In: IEEE. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2018. p. 115–126. Citado 5 vezes nas páginas 21, 23, 53, 122 e 123.

Apêndice A
TOOL REQUIREMENTS

A.1 Tool Requirements

A.1.1 Functional Requirements

- **RF1.** Retrieve the Java method from a public GitHub repository
  Input: Github repository URL; Method Name
  Description: Given a GitHub link and a target method name, the system downloads the complete class source code from the remote repository, parses it, and locates the method definition by name.
  Output: Source code of the method

- **RF2.** Compute object-oriented metrics for the Method
  Input: Source code of the target Method
  Description: Computes 21 object-oriented and complexity metrics for the candidate method using the plugin CK.
  Output: CSV file containing the set of computed metrics

- **RF3.** Parse the candidate method
  Input: Source code of the target Method
  Description: Parses the code to generate one or many fragments. The extracted fragments consisted of consecutive lines of code, respecting the boundaries of the control structure and maintaining the internal cohesion of the segment and integrity. Additionally, two rules were applied to eliminate less useful fragments: (1) slices

whose number of lines was equal to or greater than that of the original method were discarded, and (2) fragments containing reserved keywords such as break or continue were excluded (when the entire structure is not part of the block).
Output: partial JSON file containing the original method and the fragments, following format.

OutputJSON = ⟨ methodCode : String, sourceCodeFragment : String ⟩

Where:

methodCode:	Contains the complete code of the Method
sourceCodeFragment:	Contains the extracted fragment from the Method

- **RF4.** Calculate the *totalUsageOutsideSlice* metric for each fragment
  Input: The fragments candidates, this means the code fragments extracted from the Method
  Description: A metric that indicates how many times the variables used in a fragment are also used outside that fragment within the same method. It measures the dependency of the fragment on the rest of the method: the lower the value, the more independent the fragment. The control variable (e.g. i or j) used in the *For* Loop is not considered.
  Calculation:

$$\textbf{totalUsageOutsideSlice} = \sum_{v \in \text{slice}} \Big(\text{totalInMethod}(v) - \text{inFragmentCount}(v) \Big)$$

Where:

v	Each variable present in the fragment
totalInMethod(v) :	Total occurrences of the variable in the method
inFragmentCount(v):	Occurrences of the variable inside the fragment

Output: Final JSON file containing the original method, the fragment and the totalUsageOutsideSlice metric.

OutputJSON = ⟨ methodCode : String, sourceCodeFragment : String, totalUsageOutsideSlice: Int ⟩

Where:

methodCode:	Contains the complete code of the Method
sourceCodeFragment:	Contains the extracted fragment from Method
totalUsageOutsideSlice:	sum of the use of variables outside the fragment

- **RF5.** Predict the refactoring need using a trained Random Forest model
  Input: CSV file containing the computed metrics of the Method
  Description: Uses a trained Random Forest model to predict whether the method should be refactored. The Random Forest acts as the decision gateway for the entire process: if the model predicts that refactoring is needed, the process continues; otherwise, a message is displayed to the user indicating that no refactoring is necessary.
  Output: Random Forest prediction score

- **RF6.** Generate feature-based explainability
  Input: CSV file containing the metrics, Consensual Explainer and trained Random Forest model
  Pre-condition: Random Forest score ≥ 0.5
  Description: Generates explanations based on the most relevant features, indicating why the method should be refactored. This is accomplished using the Consensual Explainer module. For more details on the internal workings of this module, see Chapter 5.
  Output: Feature-based Explanation, along with its value, weight and agreement

- **RF7.** Prepare the formatted input for the CodeBERT model
  Input: JSON file (methodCode, sourceCodeFragment, totalUsageOutsideSlice)
  Pre-condition: Random Forest score ≥ 0.5
  Description: The code is preprocessed by cleaning it (removing comments, blank spaces, etc.) and, most importantly, removing the entire fragment from the method. This step is performed to prevent information leakage.
  Output: Formatted JSON file

- **RF8.** Identify the most relevant code fragment using CodeBERT Model
  Input: JSON file (methodCode, SourceCodeFragment, totalUsageOutsideslice),

and Trained CodeBert model
Pre-condition: Random Forest score ≥ 0.5
Description: Each JSON file contains several records, each one correspondes to a different fragment extracted from the same method. To identify the best fragment, the <methodCode, SourceCodeFragment> pair is passed through the trained CodeBERT model to calculate the *Affinity* score, which measures the fragment's similarity to the rest of the Method code. A higher score indicates lower *affinity*, making the fragment a better candidate for refactoring. At the end of the iteration, a score is obtained for each fragment. The best fragment is then selected by first selecting the two fragments with the highest scores, and using the variable *totalUsageOutsideSlice* to break any ties. The score calculated by the model is normalized to a value between zero and one, providing a probability that is more intuitive to interpret.
Output: the best Code fragment selected, along with its probability

- **RF9.** Formulate the complete refactoring recommendation
  Input: Selected code fragment, feature-based explanation, list of categories (motivations for applying Extract Method), W3B criterion, and prompt
  Description: The *complete* refactoring recommendation integrates multiple elements: the selected code fragment and its associated probability value serve as the primary focus, while the feature-based explanation provides the rationale for refactoring the method. The W3B criterion is applied as a guideline for structuring a comprehensive recommendation, supported by a generic list of categories describing the potential benefits of applying the Extract Method. An elaborated prompt is then constructed to combine these inputs, highlight the identified benefits, and suggest practical details—such as a possible name for the new method and guidance on how to apply the refactoring. Finally, OpenAI GPT, an external API service, assembles and refines the complete recommendation into clear, developer-friendly natural language.
  Output: The final output includes: (1) The fragment to be refactored and its probability value; (2) Explanation based on feature relevance; (3) Textual explanation supporting the need for refactoring; (4) Benefits of applying the Extract Method; (5) Actionable guidance for implementation.

Output: Structured refactoring recommendation ready for presentation to the developer.

- **RF10.** Present the result in a web interface
  Entry: Structured Refactoring recommendation text
  Description: Displays results in a user-friendly interface, allowing developers to receive the complete refactoring recommendation in an accessible format.
  Output: Rendered HTML interface with the complete refactoring recommendation.

A.1.2 Non-Functional Requirements

- **RNF1.** Local execution with optional GPU acceleration for CodeBERT inference
  Description: The system must be able to run entirely in a local environment, leveraging GPU acceleration when available to improve inference performance for the CodeBERT model. GPU usage should be automatically detected and utilized without requiring manual configuration.

- **RNF2.**Modularity and extensibility
  Description: System components must be loosely coupled and organized to facilitate maintenance, updates, or replacement.

A.1.3 Constraints

- **C1.** Internet connectivity requirements
  Description: The tool requires an active internet connection for: i)Retrieving Java source code from GitHub repositories; and ii) Invoking the GPT-based API.

- **C2.** Public repository limitation
  Description: Only public GitHub repositories are supported for direct source code retrieval.

A.1.4 Package Structure

The project is organized into multiple packages that reflect the main responsibilities of the application.

- **app/** – Contains the FastAPI application code, including endpoints, routes, and utilities for communication between the front-end and the models.
- **best_model/** – Stores the trained CodeBERT model, ready to be loaded and used for predictions.
- **consensus_explanability/** – Implements the explainability module, responsible for generating feature-based explanations to understand the model's decisions.
- **GPT/** – Contains scripts and utilities that interact with GPT to assist in formatting explanations and enriching the recommendations.
- **java/** – Stores the FME (Fragment Metrics Extractor) Java JAR, used to extract code fragments from methods and calculate code metrics.
- **logs/** – Directory for storing application log files, recording errors, prediction executions, and system activities.
- **model/** – Contains scripts for loading the CodeBERT model and executing prediction scoring.

A.2 Implementation

A.2.1 Languages and Libraries

The Refactoring Recommendation Tool is implemented primarily in Python 3.10. The environment is managed using Conda, with additional Python packages installed via pip. Key libraries and frameworks used in the project include:

- **FastAPI** (0.115.14) – for building the backend REST API.
- **Uvicorn** (0.35.0) – ASGI server to run the FastAPI application.
- **Transformers** (4.53.0) – for loading and using the pretrained CodeBERT model.
- **PyTorch** (2.7.1) – for model execution and inference.
- **Scikit-learn** (1.7.0) – for the Random Forest classifier used in code analysis.

- **Pandas** (2.2.2) and **NumPy** (1.26.4) – for data manipulation and numerical operations.
- **Matplotlib** (3.9.2), **Plotly** (5.24.1), **Kaleido** (0.2.1) – for visualization of results and explanations.
- **LIME** (0.2.0.1), **SHAP** (0.46.0), **Anchor_Exp** (0.0.2.0) – for model interpretability and explainability.
- **OpenAI** (1.93.0), **Requests** (2.32.4), **Python-Dotenv** (1.1.1) – for communication with external services.
- **Jinja2** (3.1.6) – for HTML templating in the web interface.
- **IPython** (8.27.0), **IPykernel** (6.29.5) – for interactive experimentation in Jupyter Notebooks.
- **Img2pdf** (0.5.1), **Joblib** (1.4.2) – for auxiliary functionalities such as saving figures and model persistence.
- **Jsonschema** (4.24.0), **Pydantic** (2.11.7) – for data validation and structured configuration.
- **Notebook** (7.4.4), **JupyterLab** (4.4.4) – for interactive development and experimentation.
- **Tenacity** (9.1.2) – for retry logic in API calls.
- **Spacy** (3.8.7), **Sympy** (1.14.0) – for parsing, symbolic computation, and code analysis tasks.

This environment ensures reproducibility and simplifies the installation of dependencies across different machines, covering backend development, model execution, explainability, visualization, and interactive experimentation.

A.3 Local Execution

This section summarizes the procedure for running the Refactoring Recommendation Tool in a local environment.

Pre requisites. To run the system locally, ensure that the following requirements are met:

- Install **Python 3.11** and **Java 21 (LTS)** on the local machine.
- Install the Python dependencies listed in `requirements.txt` using:
  ```
  pip install -r requirements.txt
  ```
- Configure the OpenAI GPT API key by creating a `.env` file and adding the variable:
  ```
  OPENAI_API_KEY=your_api_key_here
  ```
  This key is required for generating complete refactoring recommendations.

Step-by-Step Execution. Follow the steps below to start the tool locally:

- Launch the local server from the project directory with:
  ```
  uvicorn app.main:app --reload
  ```
- Open a web browser and navigate to `http://localhost:8000`.
- Use the input form to provide the GitHub repository URL and the method name to be analyzed.
- Submit the request and wait for the analysis to complete.

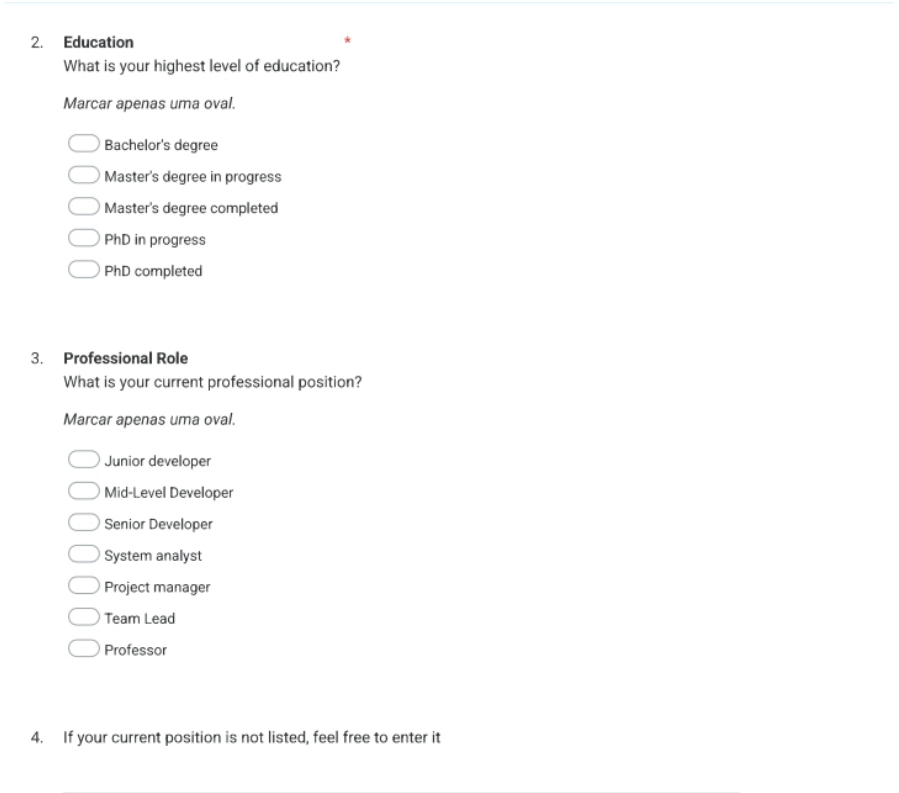
Limitations. Currently, the tool has the following limitations:

- Since only the method name is accepted as input, in cases where multiple overloaded methods share the same name but have different signatures, the tool will not process any of them.
- Methods involving the creation of anonymous classes or subclasses are not identified or fragmented by the Fragment Method Extractor (FME).

Apêndice B

PROFILE FORM AND RESULTS

B.1 Profile Form



2. **Education** *
What is your highest level of education?

Marcar apenas uma oval.

- Bachelor's degree
- Master's degree in progress
- Master's degree completed
- PhD in progress
- PhD completed

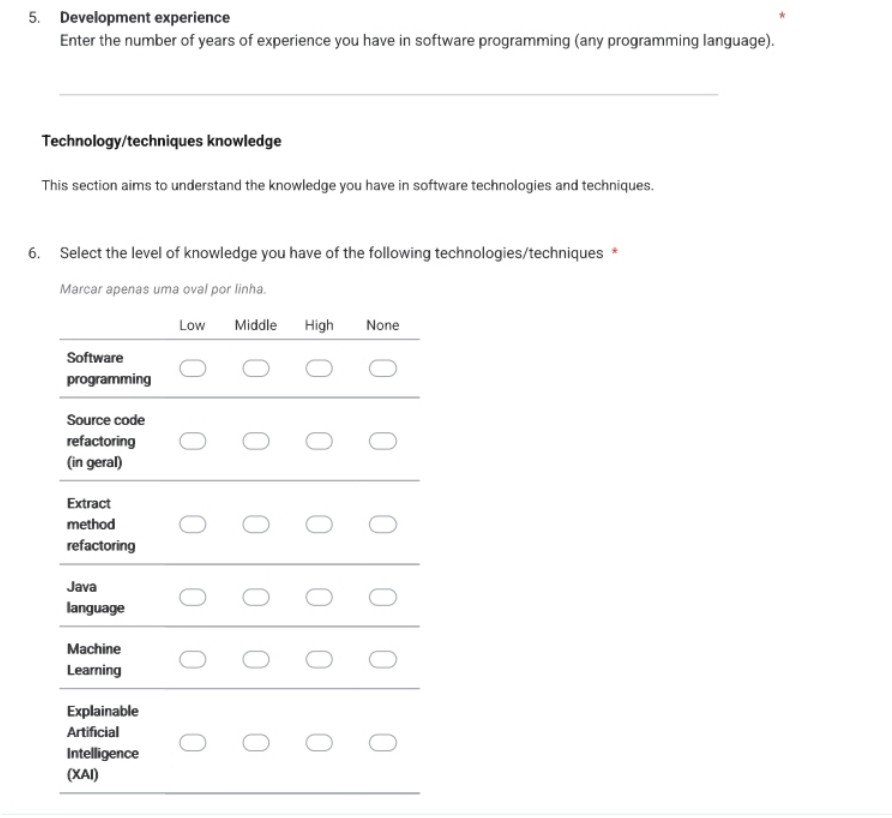
3. **Professional Role**
What is your current professional position?

Marcar apenas uma oval.

- Junior developer
- Mid-Level Developer
- Senior Developer
- System analyst
- Project manager
- Team Lead
- Professor

4. If your current position is not listed, feel free to enter it

Figure 26 – Profile Form - Questions 1-4



5. **Development experience** *
Enter the number of years of experience you have in software programming (any programming language).

Technology/techniques knowledge

This section aims to understand the knowledge you have in software technologies and techniques.

6. Select the level of knowledge you have of the following technologies/techniques *

Marcar apenas uma oval por linha.

	Low	Middle	High	None
Software programming				
Source code refactoring (in geral)				
Extract method refactoring				
Java language				
Machine Learning				
Explainable Artificial Intelligence (XAI)				

Figure 27 – Profile Form - Questions 5-6

B.2 Results

ID	Q2	Q3	Q4	Q5	Q6.a	Q6.b	Q6.c	Q6.d	Q6.e	Q6.f
P01	Master's degree in progress	Senior Developer		7	High	High	Middle	Low	Low	Low
P02	Bachelor's degree	Junior developer		2.5	High	High	High	**High**	Middle	Low
P03	Bachelor's degree	Team Lead		15	High	Middle	High	**High**	Low	Low
P04	PhD in progress	Mid-Level Developer		3	Middle	Middle	Low	Middle	Low	Low
P05	Master's degree in progress		QA	5	Middle	Middle	Middle	Middle	Low	Low
P06	Master's degree in progress	Team Lead	Cientista de Dados	10	Low	Low	Low	Low	Low	Low
P07	Bachelor's degree		QA Analyst Senior	10	Middle	Middle	Low	Low	Low	None
P08	Master's degree in progress		Quality Assurance Engineer	6	Middle	Middle	Middle	**High**	Middle	None
P09	Master's degree in progress	Mid-Level Developer		5	High	Middle	Middle	**High**	None	None

Figure 28 – Profile Form Result

B.3 Experiment Questions

Q#	Type	Dependency
Phase 1 — Without Tool Support		
Q1	Closed-ended	None
Q2	Open-ended	Q1
Q3	Closed-ended	Q1 = "Yes"
Q4	Open-ended	Q3 = "Yes"
Q5	Open-ended	Q1 = "Yes"
Q6	Closed-ended	Q5
Q7	Closed-ended	Q6 = "No"
Q8	Closed-ended	Q1 = "No"or "I'm not sure / Can't decide"
Q9	Open-ended	Q8
Q10	Likert	None
Phase 2 — With Tool Support		
Q11	Closed-ended	Q8 = "No"or "I'm not sure / Can't decide"
Q12	Open-ended	Q11
Q13	Closed-ended	Q5 answered AND (Q7 = "No"or "I'm not sure / Can't decide")
Q14	Open-ended	Q13
Q15	Closed-ended	Q11 and/or Q13
Q16	Closed-ended	None
Q17	Closed-ended	None
Q18	Open-ended	Q17
Q19	Likert	None
Q20	Likert	None

Table 25 – Question types and dependencies