Titre: Title:	Refactoring with LLMs: Insights from Martin Fowler's Catalog
Auteur: Author:	Yonnel Chen Kuang Piao
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Chen Kuang Piao, Y. (2025). Refactoring with LLMs: Insights from Martin Fowler's Catalog [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/65545/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/65545/
Directeurs de recherche: Advisors:	Foutse Khomh
Programme: Program:	GÉNIE INFORMATIQUE

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Refactoring with LLMs: Insights from Martin Fowler's Catalog

YONNEL CHEN KUANG PIAO
Département de génie informatique et génie logiciel

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

Mai 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Refactoring with LLMs: Insights from Martin Fowler's Catalog

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

Mohammad HAMDAQA, président
Foutse KHOMH, membre et directeur de recherche
Zohreh SHARAFI, membre

DEDICATION

To everyone who has supported me on this journey...

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincerest gratitude to my research director, Foutse Khomh, for his invaluable guidance, continuous encouragement, insightful feedback, and financial support. His expertise and unwavering belief in my ability to complete this thesis have been crucial in helping me navigate the challenges I faced along the way. Despite his demanding schedule, he consistently made time to offer me essential feedback and support.

I would also like to extend my heartfelt gratitude to my contributors, Arghavan Moradi Dakhel and Leuson Da Silva, for their exceptional leadership and invaluable contributions throughout the course of this thesis.

Finally, I wish to to express my sincere appreciation to the members of the jury, Professor Zohreh Sharafi and Professor Mohammad Hamdaqa for kindly agreeing to evaluate this thesis.

RÉSUMÉ

La refactorisation du code est une pratique essentielle en génie logiciel qui consiste à restructurer du code existant pour améliorer sa structure interne, sa lisibilité et sa maintenabilité, sans altérer son comportement ou ses fonctionnalités externes. L'objectif principal est d'améliorer la qualité du code, de réduire sa complexité et de faciliter les modifications futures, rendant ainsi le code plus efficace et évolutif au fil du temps. Toutefois, malgré son rôle crucial dans l'assurance de la qualité et de la maintenabilité à long terme du code, le processus de refactorisation est souvent négligé par les développeurs. Ce phénomène est principalement causé par la nature chronophage de cette pratique, ainsi que par ses importantes exigences en termes d'efforts et de ressources, ce qui en fait souvent une tâche difficile et de faible priorité dans des contextes de cycles de développement rapides.

Même si de nombreux outils automatisés, comme des plugins dans les environnements de développement intégrés (IDE, de l'anglais *Integrated Development Environments*), ont émergé au cours des dernières années pour alléger une partie de la charge de travail, ces outils sont souvent limités dans leur capacité à traiter une large gamme de types de refactorisation. En effet, ils se concentrent généralement uniquement sur des refactorisations simples et couramment utilisées.

L'émergence des grands modèles de langage (LLMs, de l'anglais *Large Language Models*) ces dernières années a suscité un intérêt considérable dans le domaine du génie logiciel, notamment en raison de leurs capacités avancées à résoudre des problèmes. Ces modèles ont déjà été utilisés pour évaluer la qualité du code, ainsi que pour effectuer des réparations automatisées du code. Dans ce mémoire, nous étendons l'application des LLMs au domaine de la refactorisation du code, en étudiant leur capacité à effectuer des tâches de refactorisation. Pour atteindre cet objectif, ce mémoire est divisé en deux sous-objectifs.

Dans le premier objectif, nous étudions l'efficacité des LLMs dans l'application d'une large gamme de types de refactorisation. Dans le deuxième objectif, nous examinons comment différents niveaux de contexte influencent la capacité des LLMs à effectuer des tâches de refactorisation.

Nous commençons par extraire des informations clés provenant du catalogue de refactorisation de Martin Fowler, telles que les noms des types de refactorisation, ainsi que les instructions détaillées pour les appliquer, afin de concevoir et d'évaluer la performance de cinq types de requêtes différentes (Basées sur aucun exemple, Basées sur deux exemples, Contextuelles, Basées sur des instructions et Basées sur une règle) pour effectuer 61 types de refactorisation

en utilisant GPT-4o-mini, ainsi que DeepSeek-V3. Pour garantir une évaluation complète, nous utilisons deux ensembles de données : les exemples contenus dans le catalogue de Martin Fowler et des projets GitHub, qui couvrent une large variété de types de refactorisation et des extraits de code réels. Nous évaluons la performance des LLMs dans ces tâches de refactorisation à travers l'évaluation de la préservation sémantique, ainsi que de divers indicateurs de qualité du code, afin d'évaluer l'efficacité de ces différents types de requêtes.

Nos résultats suggèrent que, bien que les instructions détaillées soient généralement plus intuitives pour les humains dans le cadre de tâches de génie logiciel telles que la refactorisation, les requêtes formulées selon des règles surpassent les autres types de requêtes, y compris celles basées sur les instructions, lorsqu'elles sont utilisées avec le modèle GPT-4o-mini. De plus, DeepSeek a montré une performance supérieure en présence d'une quantité réduite d'informations contextuelles, alors que les requêtes basées sur le contexte et celles ne comportant aucun exemple ont montré de meilleurs résultats comparativement aux autres types de requêtes.

ABSTRACT

Code refactoring is a fundamental practice in software engineering that involves restructuring existing code to improve its internal structure, readability, and maintainability, without altering its external behavior or functionality. The primary goal is to enhance code quality, reduce complexity, and facilitate future modifications, making software more efficient and adaptable over time. However, despite its critical role in ensuring long-term code quality and maintainability, the refactoring process is often neglected by developers. This is primarily due to the substantial time, effort, and resources it demands, often making it a challenging and low-priority task in fast-paced development cycles.

While numerous automated tools, including plugins in Integrated Development Environments (IDEs), have emerged in recent years to alleviate some of the workload, they are often limited in their ability to address a wide range of refactoring types, typically focusing only on more basic and commonly used refactorings.

The emergence of Large Language Models (LLMs) in recent years has sparked significant interest in the field of software engineering, particularly due to their advanced problem-solving capabilities. These models have already been utilized to assess code quality and perform automated code repairs. In this thesis, we extend the application of LLMs to the domain of code refactoring, investigating their effectiveness in handling a broad range of refactoring tasks. To accomplish this objective, the thesis is structured around two sub-objectives.

In the first objective, we investigate the effectiveness of LLMs in applying a wide range of refactoring types. In the second objective, we examine how varying levels of context influence the ability of LLMs to perform refactoring tasks.

We start by extracting key insights from Martin Fowler's catalog of refactorings, such as refactoring names and associated step-by-step instructions, to design and evaluate the performance of five distinct prompt types (*Zero-shot*, *Two-shot*, *Context-based*, *Instruction-based*, and *Rule-based*) in performing 61 refactoring types using GPT-4o-mini and DeepSeek-V3. To ensure a comprehensive evaluation, we utilize two datasets—Martin Fowler's catalog and GitHub projects—that encompass a wide variety of refactoring types and real-world code snippets. We assess the performance of LLMs in refactoring tasks by evaluating semantic preservation post-refactoring, along with various code quality metrics, to gauge the effectiveness of different prompts.

Our findings suggest that, while step-by-step instructions tend to be more intuitive for humans in software engineering tasks like refactoring, *Rule-based* prompts outperform other prompt types, including *Instruction-based* prompts, when used with GPT-4o-mini. Furthermore, DeepSeek demonstrated superior performance with less contextual information, as the *Context-based* and *Zero-shot* prompts yielded greater success.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

LLMs	Large Language Models
LOC	Lines of code
CC	Cyclomatic complexity
FOUT	Number of method calls
GPT	GPT-4o mini
DeepSeek	DeepSeek-V3
IDEs	Integrated Development Environments

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Throughout the process of software development, systems are not only built but also continuously maintained and updated to meet evolving stakeholder demands, while simultaneously addressing any reported issues or bugs. As these systems undergo constant evolution, it becomes increasingly important to adopt practices that not only facilitate their maintenance but also ensure a seamless and efficient progression over time. Among these practices, code refactoring stands out as a fundamental and indispensable activity in software engineering [1]. This practice plays a crucial role in the health of software systems, particularly in the context of rapid, continuous development cycles. In such fast-paced environments, where new features must be delivered quickly and deadlines are often tight, there is a high risk that poor or inefficient code may be deployed to production [2]. Refactoring, therefore, becomes not just a valuable but a necessary practice to prevent technical debt and ensure the long-term sustainability of the software.

While there exists a variety of refactoring types in software engineering, Martin Fowler [3] has provided a comprehensive framework for understanding and applying these techniques. In his seminal work, he systematically categorized refactoring into 61 distinct types, compiling them into an extensive guidebook. This book not only offers in-depth explanations of each refactoring type but also provides clear motivations behind each practice, step-by-step instructions for their implementation, and numerous illustrative code examples to aid developers in understanding the nuances between each of them. Fowler's catalog has proven to be a cornerstone in the field, serving as a critical reference point for many previous research efforts and real-world refactoring applications [4–9].

1.1 Definitions and basic concepts

1.1.1 Code Refactoring

Code refactoring in software development is the act of restructuring an existing body of code by making small, behavior-preserving changes to improve its internal structure without altering its external behavior. Its primary goals are to increase readability, maintainability, and quality, while preserving its semantic integrity [3]. Here is a simple Python example of *Rename Method* as per Martin Fowler [3]:

```
def calculate(a, b):
    return a + b
```

↓

```
def add_numbers(a, b):
    return a + b
```

The process of applying code refactoring involves a series of methodical and critical steps. Initially, developers must identify a code segment that requires refactoring, which often arises from areas of complexity, redundancy, or poor readability. Once the target code segment is identified, the next step is to carefully select the most appropriate refactoring type, considering the specific issue at hand and the desired outcome. After choosing the right refactoring strategy, developers proceed to implement the change, ensuring that the code is modified in a way that enhances its readability, maintainability, or performance. Throughout this process, it is imperative to ensure that the code's external behavior remains intact. Each of these steps must be carefully executed to maintain the balance between improving code quality and preserving system functionality [3].

1.1.2 Large Language Models

Natural Language Processing (NLP) is a branch of artificial intelligence (AI) that aims to enable machines to comprehend and interact with human language. Historically, language modeling has evolved through four developmental stages:

1. (1990-2000) Statistical Language Models (SLM) emerged in the 1990s with the development of statistical methods that estimate the likelihood of a sentence appearing within a given text.

2. (2000-2017) The limitations of SLMs led to the development of Neural Language Models (NLMs), which utilize neural networks to process significantly longer word sequences and capture more complex patterns.

3. (2017-2020) The subsequent phase in the evolution of language modeling was marked by the emergence of Pre-trained Language Models (PLMs). These models are initially pre-trained through unsupervised learning on large-scale text corpora, allowing them to

develop a better understanding of fundamental language structures. After pre-training, these models can be fine-tuned on smaller, more specific datasets to perform specialized tasks.

4. (2020 - Present) Building upon PLMs, Large Language Models (LLMs) are sophisticated neural network models trained on vast text datasets to comprehend and generate human-like text. They are distinguished by their significant growth in model size, dataset scale, computational power, and ability to utilize contextual information. Moreover, they have demonstrated exceptional adaptability compared to PLMs, evolving from task-specific models to versatile, general-purpose models [10].

1.2 Motivation

Traditionally, code refactoring has been a manual process carried out by practitioners who rely on industry best practices and their experience to guide their decisions.

However, despite its necessity and numerous benefits, developers often hesitate to engage in this process, primarily due to its time-consuming and error-prone nature [2]. As a result, both researchers and practitioners have sought to automate key aspects of the refactoring process to alleviate some of these challenges. This has led to the development of various assistant tools designed to help developers perform refactoring tasks more efficiently. Over time, these efforts have contributed to the integration of refactoring plugins into Integrated Development Environments (IDEs), such as Eclipse, which support a limited set of refactoring types [11–13].

However, Eilertsen et al. [14] reported that developers often avoid using these refactoring tools. The authors argue that, among other reasons, while these tools are effective at handling simpler refactorings, they tend to struggle with more complex ones and are limited in their ability to support a broader range of refactoring types.

Several studies have focused on developing tools that automatically detect refactorings, with some of these tools already being used by developers in software projects, such as RefactoringMiner [15] and RefFinder [16]. Although these tools cannot predict the appropriate refactoring type needed for a given code snippet or apply it automatically, they provide valuable insights by identifying the refactorings that developers have implemented across a wide range of software projects.

In recent years, rapid advancements in LLMs have sparked significant interest in leveraging their capabilities to support various software development tasks. Given that LLMs are trained on vast datasets consisting of human-written code and natural language, they have proven

to be strong candidates for automating numerous tasks within the software development lifecycle. Among these, one area that has garnered attention is the application of LLMs in automating code refactoring [2, 17–19].

For instance, Liu et al. [17] introduced RefBERT, a tool specifically designed to automate one type of refactoring: renaming. RefBERT aims to enhance code clarity by focusing on renaming variables and methods, ensuring that the names are more descriptive and aligned with the intended functionality of the code.

Similarly, Cordeiro et al. [20] explored the use of LLMs to refactor open-source Java projects, primarily targeting the reduction of code smells and the improvement of overall code quality. Their approach, however, did not focus on specific refactoring types or best practices. The results of their study revealed that, even in the absence of explicit guidance or detailed instructions, LLMs were able to successfully refactor code, particularly for more straightforward refactoring types, such as *Rename Method* and *Extract Method*. These simpler refactorings are known to be easier to detect and implement due to their relatively low complexity [20]. However, the findings also suggest a limitation: LLMs tend to perform better on these simpler refactoring types, possibly due to their limited understanding of the broader spectrum of refactoring best practices commonly employed by developers in real-world scenarios. This highlights a critical gap in current research and tools, underscoring the need for more sophisticated and versatile LLM-based refactoring tools capable of handling a wider variety of refactoring types, including more complex ones that require a deeper understanding of both code quality and developer intent.

1.3 Thesis statement

Aiming to address the gaps highlighted in existing research, this thesis seeks to extend the application of LLMs to automatically perform a broader range of refactoring types. Specifically, we investigate how we can guide LLMs in learning to apply various code refactorings, harnessing their in-context learning capabilities. Our approach is inspired by established software engineering guidelines, such as Martin Fowler's comprehensive refactoring catalog [3].

This thesis focuses on three key steps within the refactoring process: (i) the application of refactorings, (ii) the evaluation of their impact on overall code quality, and (iii) ensuring the preservation of semantic integrity throughout the process. To conduct our investigation, we select two distinct LLMs for comparison: OpenAI's closed-source model GPT-4o-mini and the open-source model DeepSeek-V3. To enrich the context provided to these models, we incorporate various types of information drawn from Fowler's catalog, enabling us to explore

how different contextual elements influence the models' performance in refactoring tasks.

Overall, our investigation is structured around the following key objectives:

- 1. Examine the effectiveness of LLMs in applying a diverse set of refactoring types.
- 2. Explore how various types of contextual information can influence and guide LLMs in performing refactoring tasks.

1.4 Thesis Outline

The rest of the thesis includes material adapted from my paper of the same title and is structured as follows:

- Chapter 2 provides a review of related work in the field of code refactoring and the application of LLMs in software engineering tasks.
- Chapter 3 presents a research article detailing our methodology and results.
- Chapter 4 concludes the thesis, discusses the limitations of our work, and suggests directions for future research.

CHAPTER 2 LITERATURE REVIEW

This chapter provides a comprehensive review of the literature related to the topic of this thesis. In Section 2.1, the focus is on the benefits and advantages of code refactoring, highlighting its significance in improving software quality and maintainability. Section 2.2 examines various tools employed for code refactoring and analysis that do not rely on LLMs, providing an understanding of traditional approaches in this domain. Section 2.3 explores the emerging role of LLMs in evaluating code quality and facilitating automatic code repair, shedding light on their potential to revolutionize software maintenance. Section 2.4 delves into recent advancements where LLMs are specifically applied to code refactoring tasks, exploring their effectiveness and the challenges they present. Section 2.5 discusses how LLMs are being utilized across a broader spectrum of software engineering tasks, offering insights into their versatility and growing influence in the field.

2.1 Code Refactoring

Al Dellal and Abdin [21] conducted a systematic literature review to analyze empirical evidence on the impact of object-oriented code refactoring on software quality attributes, synthesizing results from 76 primary studies. The review found that, while refactoring techniques generally improve internal quality attributes, such as cohesion and complexity, their impact on external quality attributes, such as flexibility and maintainability, is more variable. For example, refactorings like *Extract Class* and *Extract Method* were shown to reduce cyclomatic complexity by up to 30% and improve cohesion by approximately 15%. However, these refactoring techniques can sometimes lead to trade-offs, such as increased coupling or negative impacts on maintainability, with certain studies reporting up to a 10-15% increase in coupling in some scenarios. Additionally, the paper highlighted that about 62% of studies showed a positive impact on maintainability, while a smaller proportion (12%) indicated negative effects under specific circumstances. These findings underscore the complex and context-dependent nature of refactoring, suggesting that, while it can be beneficial, its effects on software quality should be carefully assessed in each case.

2.2 Tools for Refactoring and Code Analysis

Over time, assistive tools that support practitioners in performing refactorings have proven to be highly relevant in daily programming tasks for detecting and applying refactorings.

Among these tools, COMEX [22] stands out as a framework that enables developers to generate code representations, such as Control Flow Graphs (CFG) and Abstract Syntax Trees (AST), which can be leveraged by LLMs for software engineering tasks. This approach allows LLMs to rely on structural and semantic code properties, rather than simply on an arrangement of tokens.

Another notable approach to assistive code refactoring is the use of heuristic search algorithms. In this context, the tool Opti Code Pro [23] leverages this approach to guide the refactoring process, assessing potential modifications through predefined heuristics. By implementing a best-first search algorithm, the tool is guided by heuristic functions that evaluate the impact of refactorings on key software quality metrics. The results demonstrated that Opti Code Pro was effective in identifying beneficial refactoring opportunities, leading to improvements in code quality.

Furthermore, RefactoringMiner 2.0 [24] is widely used for detecting refactorings in Git repositories. In contrast to its predecessor [15], the new release supports submethod-level refactorings, such as *Extract Method* and *Rename Method*. In its evaluation, RefactoringMiner 2.0 reported high precision and recall rates (99.6% and 94%, respectively), making it one of the most reliable tools for automated refactoring detection. Similarly, Ref-Finder [16] is another tool designed to detect refactorings in Java code by analyzing structural changes within the source code. Ref-Finder operates as a rule-based tool, utilizing predefined rules to identify various refactoring types. To function, the tool requires both the pre- and post-change versions of a Java repository to identify transitions that can be classified as refactorings. Compared to other tools, Ref-Finder covers a broader range of refactoring types (63 out of the 72 types outlined in the first edition of Fowler's catalog).

Regarding tools for performing refactorings, practitioners are typically supported by IDEs, such as IntelliJ IDEA [25], Eclipse [26], and Visual Studio Code [27], which provide automated refactoring services either natively or through plugins. Notable examples include the JRefactory [28] and Eclim [29] plugins for Eclipse [28], as well as the native support for simple refactoring tasks provided by Visual Studio Code [30] and IntelliJ IDEA [31]. These services offer essential functionalities such as *Extract Method*, *Extract Variable*, and *Rename Variable*. However, while they are effective at applying simple and moderate refactoring types, their scope remains limited to basic code changes, and they generally fall short when it comes to supporting more complex and advanced refactoring types. As a result, developers are often left with limited support for more intricate refactoring tasks.

2.3 LLMs in Code Quality and Code Repair

The application of LLMs in software engineering has gained significant attention, particularly in the areas of code quality assessment and automated code repair.

For example, Wadhwa et al. [32] introduced CORE, a tool consisting of two LLMs: a proposer and a ranker. The proposer suggests code modifications based on recommendations from static analysis tools, while the ranker evaluates these changes using criteria similar to those employed by developers. The evaluation of CORE showed that it successfully performed code revisions across 52 quality checks for 59.2% of Python files, with modifications accepted by both a static analysis tool and a human analyst. For Java, CORE achieved a successful revision rate of 76.8% across 10 quality checks, delivering results comparable to those of specialized tools but with significantly less development effort required.

In a novel approach, Bouzenia et al. [33] introduced RepairAgent, an autonomous agent that repairs bugs by interacting with appropriate tools and dynamically adjusting its approach based on feedback from previous actions. When evaluated on the widely-used Defects4J dataset [34], RepairAgent successfully repaired 164 bugs, 39 of which could not be addressed by previous solutions. The tool also demonstrated notable efficiency, with an average cost of 14 cents per bug fix, utilizing 270,000 tokens, based on OpenAI's GPT-3.5 pricing.

Furthermore, the potential of LLMs for identifying and repairing security vulnerabilities has been explored. In a series of experiments, de-Fitero-Dominguez et al. [35] demonstrate that LLMs can effectively detect and resolve a variety of security flaws in source code, including common vulnerabilities such as SQL injection attacks and cross-site scripting. The authors also introduce improvements to the LLM-based repair process, enhancing its ability to generate fixes that adhere to best security practices while maintaining code integrity. The results underscore the potential of LLMs to significantly improve the process of securing software systems by automating vulnerability repair, reducing reliance on manual intervention, and accelerating the implementation of secure code.

Lastly, in their paper "*Evaluating Source Code Quality with Large Language Models: a comparative study*" [36], the authors present a comparative analysis between LLMs and the widely used static analysis tool SonarQube. They evaluate the ability of LLMs to detect code smells and assess the code maintainability. The results suggest that LLMs can perform competitively in detecting common code issues, while also highlighting the potential benefits of integrating these models with existing analysis tools to enhance both accuracy and comprehensiveness.

2.4 LLMs In Code Refactoring

Large Language Models have revolutionized the development of software, particularly due to their ability to generate and summarize code [37–39]. As a result, the application of LLMs in code refactoring has gained significant attention in recent years. Although this area of research is still emerging, several studies have already explored its potential and made notable advancements [2, 17–19].

Similar to our current work, Shirafuji et al. [2] utilized an earlier version of OpenAI's LLM, GPT-3.5, to produce less-complex Python code by providing the model with zero-shot, one-shot, and few-shot examples aimed at reducing code complexity. As a result, they demonstrated that 95.68% of programs could be refactored, reporting a decrease in both code complexity and lines of code for the semantically correct programs, with average reductions of 17.35% and 25.84%, respectively.

Pomian et al. [18] explore how to combine the limitations of LLMs with the static code analysis capabilities of IDEs to support the *Extract Method* refactoring. Their preliminary study, which involved a sample of 1,752 *Extract Method* scenarios using LLMs, found that up to 76.3% of the suggestions were incorrect due to hallucinations [40]. To address this issue, the authors proposed EM-Assist, an IntelliJ IDEA plugin that leverages LLMs to suggest more relevant method extractions compared to traditional IDE plugins. Their results showed that LLMs performed more successfully when contextualized by the static analysis provided by the IDE tools.

With a similar goal in mind, Liu et al. [17] proposed a two-stage framework, RefBERT, to automate the *Rename* refactoring process. The framework utilizes bidirectional encoder representations from transformers (BERT), a pre-trained model originally designed for natural language processing tasks [41]. RefBERT consists of two stages: (i) providing context using the BERT model and (ii) leveraging that context to generate a meaningful new name based on established naming conventions. The authors demonstrated that context-aware LLMs produce better results compared to scenarios where no context is provided.

While previous studies have demonstrated the efficiency of context-aware LLMs, Choi et al. [19] showed how an iterative approach can further enhance LLM performance in the context of code refactoring, specifically in terms of readability and maintainability. The authors propose a solution that first identifies the method with the highest code complexity (CC) and then applies a refactoring to reduce it. This process is repeated iteratively until a satisfactory result is achieved. Additionally, to ensure that the original behavior is preserved, regression tests are conducted, and any changes that lead to test failures are rejected. The

evaluation of their approach showed that the average CC could be reduced by up to 10.4% after 20 iterations.

This thesis expands on previous studies by exploring the application of LLMs in code refactoring, focusing on a wider variety of refactoring types. It also introduces a comprehensive evaluation of five distinct prompt strategies, providing deeper insights into the diverse capabilities of LLMs for automating the refactoring process. By considering various approaches and refactoring scenarios, this thesis seeks to provide a more nuanced understanding of how LLMs can be utilized to improve code quality and maintainability.

2.5 LLMs in Software Engineering Tasks

The application of LLMs in software engineering has gained significant attention in recent years due to their potential to improve software development processes. Models like GPT-4 [42] and Codex [38] have demonstrated significant promise in automating key tasks within software engineering, such as code generation [43], code refactoring [2,17–19], code repair [44], and documentation [45].

Fan et al. [46] conducted a comprehensive survey of LLMs in software engineering, exploring their diverse applications across different stages of the software development lifecycle. They highlighted that LLMs have been used successfully for tasks such as code generation, bug fixing, refactoring, and even performance improvement. The authors argue that LLMs represent a valuable tool for automating time-consuming tasks, thus improving developer productivity. However, they also point out several limitations, particularly the tendency of LLMs to produce inaccurate or incomplete code, a phenomenon referred to as "hallucinations". These errors can introduce significant problems in software systems, leading the authors to suggest that further improvements to LLMs are essential for increasing their accuracy and dependability. Moreover, they suggest that a hybrid approach combining traditional software engineering techniques with the capabilities of LLMs may address these challenges and create more reliable tools.

2.6 Chapter Summary

This chapter provides a comprehensive review of the literature related to software engineering tasks, such as code analysis and code refactoring, along with the tools developed over the years to support these processes.

Section 2.2 explores various tools that do not utilize LLMs for refactoring and code analysis.

These include frameworks like COMEX, as well as tools such as Opti Code Pro, RefactoringMiner 2.0, and Ref-Finder, which rely on heuristic search algorithms, rule-based methods, and Git repository analysis to identify and apply refactorings. The section also highlights IDEs like IntelliJ IDEA, Eclipse, and Visual Studio Code, which offer automated refactoring support for basic tasks, though they are limited in their ability to handle more complex refactoring scenarios.

Section 2.3 discusses the application of LLMs in assessing code quality and supporting code repair. Tools like CORE, which employ a proposer-ranker model, have shown promising results in improving code quality by suggesting and evaluating modifications. Additionally, RepairAgent autonomously repairs bugs using dynamic feedback, and LLMs have been successfully utilized in identifying and resolving security vulnerabilities, thus contributing to the overall improvement of software systems. This section underscores the potential of LLMs in automating code repair, improving security, and complementing traditional static analysis tools.

Section 2.4 explores the growing application of LLMs in code refactoring. Several studies demonstrate the effectiveness of LLMs in tasks such as code simplification, method extraction, and renaming. Tools such as RefBERT and EM-Assist combine LLM capabilities with static analysis tools, improving the accuracy and contextual relevance of code refactorings. Additionally, iterative refactoring approaches, as illustrated by Choi et al., show that LLMs can progressively enhance code readability and maintainability, leading to substantial reductions in code complexity.

In summary, this chapter highlights both traditional and LLM-based tools for code refactoring and analysis, emphasizing the growing role of LLMs in automating and improving software engineering tasks. While LLMs show great potential in automating software engineering tasks, challenges remain, such as the need for more reliable results and the integration of LLMs with existing development tools to maximize their effectiveness.

CHAPTER 3 REFACTORING WITH LLMS: INSIGHTS FROM MARTIN FOWLER'S CATALOG

This chapter is adapted from my earlier paper of the same name to maintain consistency in presenting the methodology, the results, and the discussion.

3.1 Methodology

In this section, we present the methodology used to conduct this thesis (see Figure 3.1). First, we explain how we establish the datasets of realistic and real refactorings. Next, we discuss the different prompts considered, followed by the selection of LLMs. Finally, once we generate the refactorings for the scenarios under evaluation, we present the metrics used to guide our analysis. In this section, we explain the process of collecting our datasets. As reported previously, this thesis considers two datasets that cover realistic and real scenarios of refactorings.

3.1.1 Data Collection - Refactoring Scenarios

In this section, we explain the process of collecting our datasets. As reported previously, this thesis considers two datasets that cover realistic and real scenarios of refactorings.

Realistic Scenarios - Fowler's Catalog

Martin Fowler's book aggregates a set of 61 refactoring types [3]. Each refactoring is broken down into the following template: (i) name, (ii) graphical representation, (iii) illustrative code snippet, (iv) motivation, (v) instructions guiding practitioners on how to apply the associated refactoring, and, in most cases, (vi) real code examples. Figure 3.2 presents an example of the information provided for *Extract Function*.

Considering the diversity of refactoring types provided, we decided to use his book for two purposes: (i) adopting the refactorings and associated information as a dataset of realistic refactoring scenarios and (ii) using their step-by-step procedures and code examples as input for our LLM prompt engineering (see Section 3.1.2).

For the former, we manually gathered the pre- and post-refactoring illustrative code snippets for each refactoring type found in this book from Martin Fowler's official Catalog of Refactorings website [47] (step 3 in Figure 3.2). The catalog website provides a well-organized

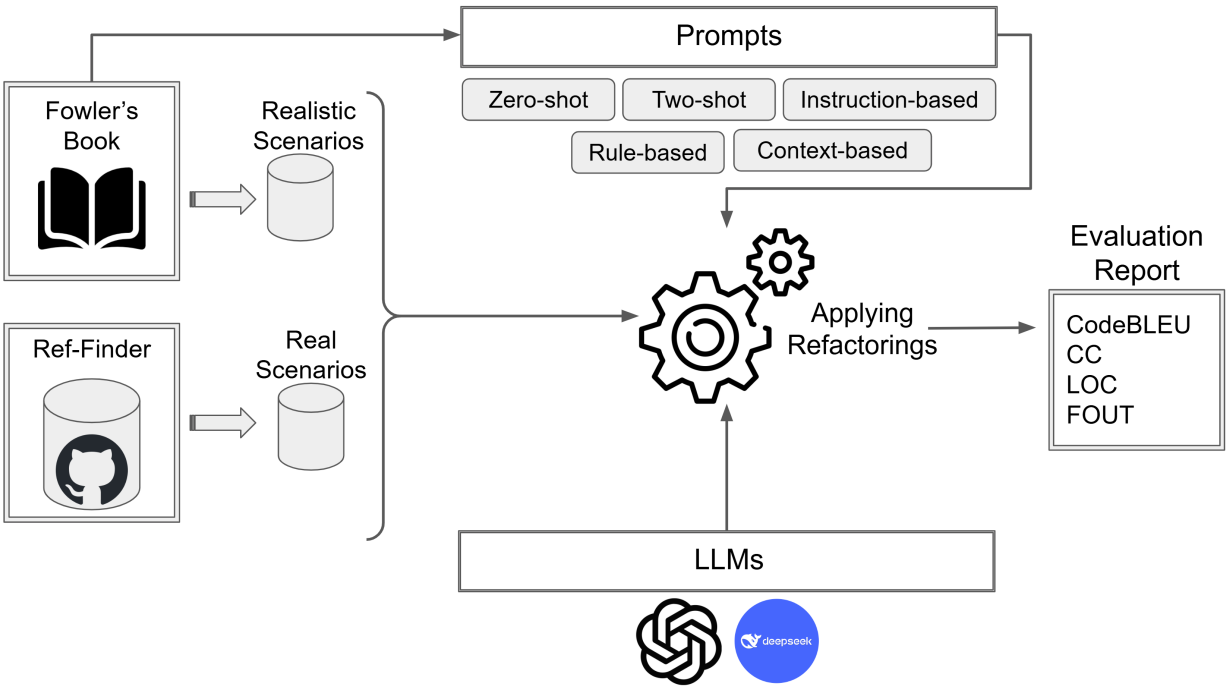

Figure 3.1 Methodology adopted for conducting this thesis.

compilation of all illustrative code snippets featured in the second edition of Fowler's book.

Finally, we consider all 61 refactoring types, which represent our dataset of realistic scenarios. Each refactoring includes a name and the associated pre- and post-refactoring code.

Real Scenarios - Ref-Finder Tool [48]

To gather real scenarios of refactorings aligned with Martin Fowler's catalog, we selected scenarios reported by Kádár et al. [48], mined from GitHub repositories using the Ref-Finder tool.

In their study, Kádár et al. [48] built their dataset using seven Java open-source projects, covering 19 method-based refactoring types. The dataset includes 19 method-level refactoring types, covering 7,872 samples, 626 of which were manually validated by the authors. To ensure the quality of our dataset, we first relied on 145 True Positive (TP) samples from the validated dataset. After carefully examining the dataset, we selected 35 scenarios, distributed across 5 method-level refactoring types, for which we could compile the pre- and post-refactoring versions (commits). During this process, we prioritized projects with the highest number of TPs, starting with ANTLR4, followed by JUnit. For the remaining projects, we were unable to execute them. To expand our dataset, we reviewed additional

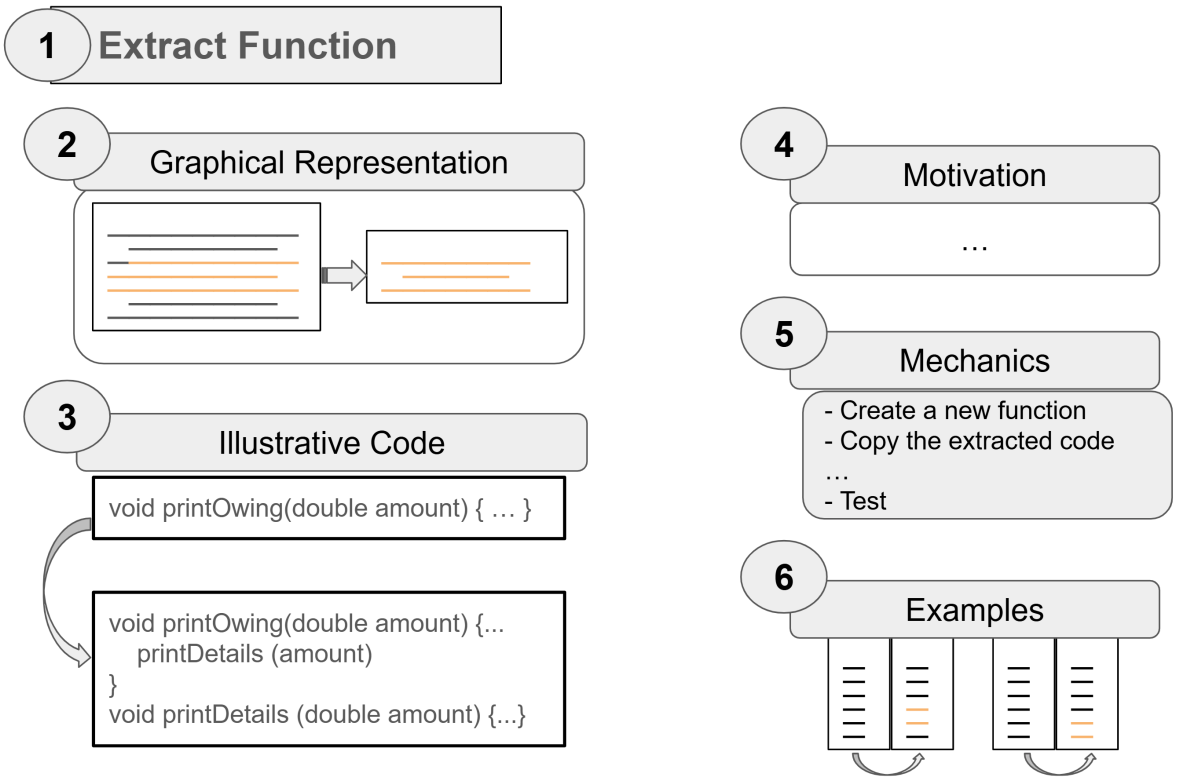

Figure 3.2 Extract Function: refactoring information collected from Fowler's Book.

examples collected reported by the original dataset that had not been manually validated. Ultimately, the thesis encompassed 53 scenarios spanning 11 distinct refactoring types. Some types, like *Extract Method*, had more correct examples, while others had very few, with some types, like *Replace Subclass with Delegate*, yielding no validated examples.

Manual Validation Process

Knowing that the previously selected dataset consists of refactorings collected using Ref-Finder and manually validated by two researchers, we decided to further validate them to ensure they were valid cases. To do so, we followed the same methodology adopted in the original study. For each refactoring, two researchers independently evaluated the samples, later discussing any disagreements and computing overall agreement using Cohen's Kappa coefficient [49]. Samples without consensus were excluded from the dataset. Notably, the dataset includes only those samples for which both validators agreed with the label assigned by Ref-Finder.

3.1.2 Prompt Engineering

Once the datasets of refactorings were collected, we proceeded to design the prompts used when calling the LLMs. As previously shown in Figure 3.2, Fowler's book provides a comprehensive and concise overview of various refactorings. Based on this information, our scripts parsed the PDF version of the book and extracted the relevant content. In our case, we focused on the name of the refactoring, the step-by-step instructions for its application, and the written code examples (steps 1, 4, and 6, respectively, in Figure 3.2).

These code examples present an initial version of the code, followed by a sequence of updates corresponding to individual steps in the refactoring process, ultimately leading to the refactored version. Below, we explain how the extracted information was used to construct the different prompts employed in this thesis.

Zero-Shot Prompt

Zero-Shot learning is a machine learning pattern in which a model is prompted to perform a task it was not specifically trained to do [50]. Following this approach, we constructed our prompt by asking the LLMs to apply a given refactoring type to a code snippet without providing any additional context. For example, in the case of the *Extract Variable* refactoring, the prompt includes only the name of the refactoring. The following template is used:

Apply the <refactoring method> refactoring on the following code to generate Java code: <code>
Clean the output to only show the final version of the
code and do not include non-programming language.

Where *<refactoring method>* represents the desired refactoring type and *<code>* refers to the code segment to be refactored.

Two-Shot Prompt

Similar to zero-shot learning, two-shot learning occurs in a context where a model is trained on a very limited dataset to perform a specific task. However, to enhance its performance, the model is provided with a small set of supervised examples—two in this case [51]. This prompt is designed to evaluate the performance of LLMs when given additional context in the form of two examples. In this thesis, we provide the LLM with two code examples: (i) the illustrative code snippet from Fowler's catalog and (ii) the first written code example presented in the book, along with the refactoring type associated with the code changes

applied (see Figure 3.2). As the illustrative code snippet serves as a learning example for our *Two-shot* prompt, this prompt was excluded for our dataset of realistic scenarios, as such information could bias the LLMs output. The following template is used:

Following are examples to apply the <refactoring method> refactoring: <refactoring examples>
Now, given the following code, apply the <refactoring method> refactoring on it to generate Java code: <code>
Clean the output to only show the final version of the code and do not include non-programming language.

Where *<refactoring method>* represents the desired refactoring type, *<refactoring examples>* represents the two examples, and *<code>* refers to the code segment to be refactored.

Instruction-based Prompt

For our *Instruction-based* prompt, we provide the LLM with a step-by-step guide on how to perform the refactoring (instruction guiding), as previously described and shown in Figure 3.2 (step 5). This approach is tailored to each refactoring type and outlines the specific steps required for its application. The prompt is designed to evaluate the performance of LLMs when provided with additional context in the form of explicit instructions. For instance, the steps for the *Extract Variable* refactoring are as follows:

- Ensure that the expression you want to extract does not have side effects;
- Declare an immutable variable. Set it to a copy of the expression you want to name;
- Replace the original expression with the new variable;
- Test.

For this prompt, the following template is used:

Following are step-by-step instructions on how to apply the <refactoring method> refactoring: <steps>
Now, given the following code, apply the <refactoring method> refactoring on it to generate Java code:<code>
Clean the output to only show the final version of the code and do not include non-programming language.

Where *<refactoring method>* represents the desired refactoring type, *<steps>* represents the set of instructions to perform the desired refactoring type, and *<code>* refers to the code segment to be refactored.

Rule-based Prompt

This prompt is based on the LSDiff rule representation introduced by Prete et al. [52]. These rules, organized into a catalog [53], describe patterns in a template-based format, which we use as input for the prompt. While these rules were based on the first edition of Fowler's Catalog of Refactorings, this thesis relies on the second edition, which could result in mismatches when comparing the rules and associated refactoring types. However, after manually reviewing each rule, we found that 46 of them could be matched to our database of 61 refactorings, leaving 15 refactoring types without an associated rule.

Finally, for each supported refactoring type, the LLM is provided with a (i) refactoring type, (ii) its associated rule, and (iii) the target code to be refactored.

Here is an example for *Rename Method* (referred to as *Change Function Declaration* in the second edition of the book). The associated rule states that a valid refactoring is observed when a method is added, another is removed, and both methods share the similar content. Below, the rule is presented:

`added_method(newmFullName, newmShortName, tFullName)` $\wedge$
`deleted_method(mFullName, mShortName, tFullName)` $\wedge$
`similarbody(newmFullName, newmBody, mFullname, mBody)` $\rightarrow$
`rename_method(mFullName, newmFullName, tFullName)`

For this prompt, the following template is used:

Following is the rule to apply the <refactoring method> refactoring: <rule>
Now, given the following code, apply the <refactoring method> refactoring on it to generate Java code: <code>
Clean the output to only show the final version of the code and do not include non-programming language.

Where *<refactoring method>* represents the desired refactoring type, *<rule>* represents the rule to perform the desired refactoring type, and *<code>* refers to the code segment to be refactored.

Context-Based Prompt

In this final prompt, no information about the target refactoring type is provided to the LLM. Instead, the LLM is given a general definition of refactoring, stating that the process aims to improve readability, maintainability, and quality without altering the initial code's external behavior. Following this, the target candidate code for refactoring is provided. The motivation behind this prompt is to evaluate whether LLMs can correctly apply the appropriate refactoring type without being explicitly instructed to do so. The following template is used:

Code refactoring is the process of changing source code for better readability, maintainability, and quality without changing its external behavior. Given the following code, output a refactored version of it in Java: <code>
Clean the output to only show the final version of the code and do not include non-programming language.

Where *<code>* refers to the code segment to be refactored.

3.1.3 LLMs Selection

To evaluate the capability of LLMs to perform refactorings, we selected two models, as described below. The first model is OpenAI's GPT-4o-mini [42], a state-of-the-art LLM designed to perform a wide variety of natural language processing tasks. This model is proficient in generating human-like text and understanding and summarizing large volumes of information. Compared to its predecessors, this iteration offers improved coherence and factual accuracy when handling complex tasks, making it an ideal candidate for our experiment. The second LLM selected for this thesis is DeepSeek's DeepSeek-V3 [54], another state-of-the-art model designed to assist users with various tasks, such as information retrieval and content generation. This model leverages large-scale datasets to deliver precise and contextually relevant responses. Its proficiency in natural language processing makes it an ideal candidate for our experiment.

To prompt these models, our scripts rely on the services they provide. We generate access keys for each model, allowing us to interact with them through their APIs. Regarding hyperparameters, we used the default settings for each model, including temperature and the maximum token window.

3.1.4 Prompting LLMs for Applying Refactorings

Once we collected all the necessary information, we proceeded with prompting the LLMs to address the refactoring scenarios under analysis. For each scenario, we prompt each LLM five times, using all applicable prompts. To illustrate this process, consider the following JSON file as input, where we can observe the refactoring type and the target code for a given scenario:

```
{
  "REALISTIC_Example_01": {
    "RefactMethod": "EXTRACT VARIABLE",
    "BeforeCode": "function printOwing(invoice) {...}"
  }
}
```

The information used to populate the different prompts is retrieved from a JSON file, which organizes the relevant data according to the refactoring type. For example, consider the following JSON file, which contains information for the *Rule-based* prompt. When prompting the LLMs to handle an *Extract Function* operation using the *Rule-based* prompt, the necessary information is extracted from this JSON file:

```
  {
  "EXTRACT FUNCTION": {
    "Rule": "added_method(newmFullName ...)",
  },
  "EXTRACT VARIABLE": {
    "Rule": "added_localvar(mFullName ...)",
  }
}
```

When prompting the LLMs, we instruct them to provide their output in a specific format. However, even when explicitly asking them to clean their output according to the specified format (keeping only the code after applying the refactoring), some processing was still required before evaluating the quality of their generated outputs.

Indeed, the LLMs return their output as a single string, even if it contains multiple methods and/or classes. This string needs to be broken down to facilitate testing for compilation and semantic preservation.

Hence, by analyzing some cases manually, we observed some patterns that allowed us to define regular expressions (RegEx) to accurately extract the required information. As a result, the output of the LLMs was broken into this format:

```
  {
  "Example_ID": {
    "RefactMethod": "EXTRACT VARIABLE",
    "ZeroShotCode": {
      "methods": ["method1", "method2"],
      "classes": ["class1"],
      "others": ["useless_text"]
    },
    "InstrucCode": {...}, ...
  }
}
```

3.1.5 Manual Validation

To evaluate the correctness of the applied refactorings for the realistic examples, we chose manual analysis. Since the examples evaluated from Fowler's book are simple cases intended to illustrate refactoring concepts, we could not assess the static and semantic aspects of the target code. For instance, consider the case of the refactoring *Change Function Declaration*. In this case, the function was renamed from `circum` to `circumference`. We only have partial access to the method's signature, without information about the required list of parameters and method implementation (body).

For this, we conducted a manual analysis involving two authors, who independently analyzed all the refactoring scenarios from a randomly selected run and assessed its success. Each analysis was performed individually, after which the authors discussed any conflicts and reached a final consensus. To measure the agreement between them, we calculated the Cohen's Kappa coefficient [49]. This process was carried out for both LLMs under evaluation. For GPT, we observed a Cohen's Kappa coefficient of 0.864, while for DeepSeek, the coefficient was 0.774. Overall, these high scores demonstrate a substantial agreement between the reviewers, indicating that their independent evaluations were largely consistent.

3.1.6 Automatic Validation

For the dataset containing real scenarios, we extend our analysis by examining the static and semantic impact of the LLMs' refactorings. Once we have the LLMs' output, we apply the reported changes corresponding to the associated refactoring and evaluate their correctness. To demonstrate the applicability of our approach, we provide the solution as a JAR file. This process can be broken down into two steps: (i) setup and initialization, and (ii) applying and validating refactorings. Below, we outline the process in detail.

Setup And Initialization

To support the reproducibility of our results, we create a fork of the original project on GitHub for each project in our sample. Then, our scripts automatically clone the project locally, and based on the report provided in Section 3.1.1, for a given refactoring, a checkout is performed on the commit associated before the refactoring.

Applying Refactorings

Based on the report provided in Section 3.1.4, we first locate the Java file associated with the refactoring. Next, we parse this file using JavaParser [1], generating an AST (Abstract Syntax Tree). With this AST, we can extract and modify any node in the Java source code. For the refactorings that occur at method level (the whole method or a line placed in a method), we use the associated method's name to locate the node associated with this method. Then, we replace such a method with the new method implementation reported by the LLM (see Section 3.1.4). For certain refactorings, the LLM may report new method implementations. In such cases, we add these new methods as nodes in the AST. For refactorings that occur at the class level, we follow the same process as before, but this time, we replace the entire class. To support the applicability of our approach, we provide the previous solution as a JAR file. Once this initial step is completed, we save all the changes and commit them to a new branch, allowing us to manage the different refactorings individually.

Validating Refactorings

After applying and saving all the required changes, we proceed to check the static and semantic impact of the LLM-generated refactorings. To do this, our scripts attempt to compile the resulting code. It is important to note that we ensure the original commits

[1] https://javaparser.org/

are compilable. This way, if the code is no longer compilable after applying the changes, we can attribute the compilation issues to the new changes. If the code is still compilable, our scripts then check the semantic impact by running the project's tests. To account for possible flakiness, we repeat this process five times. Finally, we report any tests that result in failures or errors. We also repeat this process for the commit with the ground truth (post-refactoring), generating the same report on the test results.

3.1.7 Metrics Calculation

Once we have the refactoring changes from the LLMs, we evaluate some quality aspects. To do this, we select a set of metrics that address both general aspects of source code and those specifically related to code generated by LLMs. Below, we provide further details about the selected metrics.

Regarding the code generated by LLMs, CodeBLEU [55] is an evaluation metric designed to assess the quality of model-generated code. This metric is an extension of the original BLEU [56], which is more suited for natural language. It adds weighted n-gram matching, AST similarity, and semantic data-flow similarity to better evaluate aspects of programming. Thus, a high CodeBLEU score indicates a high similarity in both syntax and semantics when compared to the reference code. This metrics is calculated in relation to the ground truth after refactoring. In addition, we compute other metrics to assess the quality of the programming code, such as the number of lines of code (LOC), cyclomatic complexity (CC), and the number of method calls (FOUT). While LOC represents the total number of lines in the source code, FOUT indicates the number of times a method is invoked. CC is a metric proposed by McCabe to assess and quantify the complexity of a given source code. It can be calculated by analyzing the control-flow graph (CFG) of the code [57]. These metrics were computed for both research objectives and automatically collected by our scripts, using two Python libraries: pyccmetrics [58] and codebleu [55].

Additionally, for the realistic scenarios presented in Section 3.1.5, we use the LLMs to assess the success of each refactoring. We report the success rate in percentage (%) for each refactoring type. This success rate can be calculated using the following formula:

$$\text{Success rate (\%)} = \frac{\text{Number of successful refactorings}}{\text{Total number of attempts}}$$

Where *Success rate* refers to the success rate for a given refactoring type, *Number of successful refactorings* is the count of successful refactorings for a given refactoring type according to the authors' agreement, and *Total number of attempts* represents the total number of times

a given refactoring type was attempted across all prompt types.

For the real scenarios of refactorings, as presented in Section 3.1.6, we compute the number of refactoring attempts that resulted in compilable code and tests with failures or errors.

When presenting our metrics (CodeBLEU, CC, LOC, and FOUT), we report them based on the average across the 5 runs, along with the standard deviation.

3.2 Results

In this section, we address the research objectives outlined in this thesis. First, we discuss the capabilities of LLMs to address a broader set of refactorings (Objective 1). Second, we explore the impact of prompt engineering on the quality aspects of code generated by LLMs for refactoring tasks (Objective 2).

3.2.1 Objective 1: LLMs' capability to apply refactorings

In this objective, we aim to explore the LLMs' capability to apply refactorings for a vast distribution of types. Table 3.1 and Table 3.2 show the success rate of each of the 61 refactoring types collected across the 3 to 4 different prompts (*Zero-Shot*, *Instruction-based*, *Context-based*, and *Rule-based*, if applicable) for the realistic scenarios. This manually validated data indicates that LLMs are generally effective at applying a wide variety of refactoring types on realistic data. Specifically, DeepSeek achieves a success rate of 66.67% and above (indicating that no more than one prompt type fails) for over 90% (55 of 61) of the refactoring types. Meanwhile, GPT achieves a similar success rate for more than 54% (33 of 61) of the refactoring types. In contrast, DeepSeek achieves a success rate below 50% for only about 3% (2 of 61) of the refactoring types, while GPT achieves a similar success rate for less than 28% (17 of 61) of the refactoring types.

Moreover, these tables show that both LLMs are generally successful on the same refactorings. In fact, 13 of the 14 refactoring types with a 100% success rate on GPT, also have a 100% success rate on DeepSeek. The remaining one, *Replace Parameter With Query*, achieves a 75% success rate instead on DeepSeek. Similarly, both models encounter difficulties with the same refactoring types. Notably, all 6 refactoring types that achieved a success rate below 50% with DeepSeek exhibited similar outcomes on GPT.

Regarding the CodeBLEU scores and the other three metrics (CC, LOC, and FOUT), we present the averaged scores across all 5 runs for the realistic scenarios in Appendix A.

The analysis of the CodeBLEU suggests that LLMs generally generate code that is syn-

tactically different from human-generated code in realistic scenarios. DeepSeek achieved a CodeBLEU score over 0.5 in less than 20% (12 of 61) of the refactoring types, while GPT only achieved such a score in less than 10% (6 of 61) of the refactoring types. These scores indicate that the generated code differs moderately from the reference code in syntax, structure and dataflow. Such a discrepancy can be attributed to the fact that, in many instances, the reference code omitted the method implementations, while LLMs typically assumed that these missing parts should be generated as part of their task.

DeepSeek and GPT demonstrate similar performance on many refactoring types. On one hand, GPT scored a CodeBLEU over 0.6 with 3 refactoring types:

Rename variable, *Replace Nested Condition Guard Clauses*, and *Split Variable*. On the other hand, DeepSeek achieved such score with 6 refactoring types: *Replace Nested Condition Guard Clauses*, *Rename Variable*, *Pull Up Constuctor Body*, *Extract Superclass*, *Extract Variable*, and *Split Variable*. The three refactoring types commonly shared between the LLMs are relatively straightforward and typically do not require context beyond the method itself for successful application, which could explain the high CodeBLEU score from both models.

Similarly, when checking the refactoring types that were the least successful, we observe the same ones shared by GPT and DeepSeek: *Decompose Conditional*, *Combine Functions Into Transform*, and *Encapsulate Variable*. However, these poor results can partially be attributed to the fact that LLMs generated additional method implementations as part of their tasks, reducing CodeBLEU considerably. Indeed, for these three refactoring types, Table 3.1 and Table 3.2 show that GPT achieved a success rate of 100%, 0% and 50%, respectively, while DeepSeek achieved a success rate of 100%, 67% and 100%.

Despite the similarity in CodeBLEU between these two models, they differ in some of the other metrics under analysis for many refactoring types (where lower scores indicate better performance). For example, one of the most successful refactoring type, *Replace Nested Condition Guard Clauses*, discussed earlier, presents similar metrics for CC and FOUT between GPT and DeepSeek. However, for this refactoring type, DeepSeek produced significantly lower LOC.[2]

When we compute the average value for each metric across our 5 runs and across all refactoring types, DeepSeek significantly outperforms GPT in terms of CodeBLEU and LOC.[3] This suggests that, in realistic scenarios and across a wide variety of refactoring types, DeepSeek generates code that exhibits greater syntactical similarity to human-generated code, while also being more maintainable due to its lower LOC.

[2]Mann-Whitney, p-value < 0.05
[3]Mann-Whitney, p-value < 0.05

For our dataset of real scenarios [48], Table A.3 presents the averages across our 5 runs for successful compilation rates and new tests with failures or errors, along with their associated standard deviations. We observed that the *Split Variable* and *Extract Variable* refactorings produced notable outcomes (2nd and 3rd, and 12th and 13th rows, respectively). DeepSeek reported successfully compilable code in 100% and 94% of the attempts, respectively, while GPT exhibited similar compilation rates of 96.7% and 91%. When compared with other refactorings, the previous cases are less complex, usually, taking just a few lines of code and less context as well.

The most significant divergence point between the two models occurred with the *Replace Nested Conditional with Guard Clauses* refactoring. For this refactoring type, GPT is more likely to produce code that compiles successfully compared to DeepSeek (5th and 18th rows), but it also leads to more failed tests and test errors. Such a trend is further observed when we average our data across all refactoring types. The low compilation rates for the remaining refactoring types suggest that the LLMs require more context, as these are more complex refactorings.

Table A.4 presents the average metrics for the real scenarios. The analysis of the CodeBLEU score of the *Extract Variable* refactoring suggests that while LLMs do not introduce compilation or tests with failures or errors, LLMs typically apply this refactoring type differently from human developers. Specifically, this refactoring reported the lowest CodeBLEU scores on both models (5th and 18th rows for GPT and DeepSeek, respectively). On the contrary, *Introduce Assertion* and *Introduce Special Case*, which produced two of the lowest average compilation rates, averaged some of the highest CodeBLEU scores across both models (1st and 2nd, and 13th and 14th rows for GPT and DeepSeek, respectively).

3.2.2 Objective 2: The impact of prompt engineering on code quality

While Objective 1 aimed to evaluate the LLMs' capability to successfully apply a wide range of refactoring types, Objective 2 focuses on investigating the quality of the LLM-generated code and the impact of prompt engineering on its output.

Table 3.3 presents the average metrics collected across five runs of the realistic scenarios from Fowler's dataset, aggregated by prompt type, along with the associated standard deviations. Overall, compared to the ground truth scores (3rd row), all evaluated metrics showed deterioration, as GPT- and DeepSeek-generated code consistently resulted in higher values across all prompt types (4th–11th rows). This increase in CC, LOC, and FOUT suggests that the generated code deviates further from the original quality, negatively impacting maintainability and readability. Furthermore, the fact that these metrics are considerably higher than the

Table 3.1 GPT-4o-mini: Success Rate Per Refactoring Type For Realistic Scenarios (Fowler's Dataset)

LLM	Success Rate	Refactoring Types
GPT-4o-mini	1	Combine functions into class, Consolidate conditional expression, Decompose conditional, Encapsulate record, Extract variable, Pull up field, Pull up method, Replace inline code with function call, Replace nested conditional with guard clauses, Replace parameter with query, Remove dead code, Rename variable, Split variable, Substitute algorithm
	0.75	Collapse hierarchy, Extract class, Extract function, Extract superclass, Inline function, Inline variable, Introduce parameter object, Parameterize function, Push down field, Remove flag argument, Replace constructor with factory function, Replace conditional with polymorphism, Replace superclass with delegate
	0.67	Move statements into function, Move statements to callers, Rename field, Replace loop with pipeline, Replace subclass with delegate, Split loop
	0.5	Change reference to value, Encapsulate collection, Encapsulate variable, Hide delegate, Inline class, Introduce special case, Pull up constructor body, Remove setting method, Replace primitive with object, Replace type code with subclasses, Separate query from modifier
	0.33	Replace derived variable with query, Split phase
	0.25	Change function declaration, Introduce assertion, Remove subclass, Replace function with command, Replace temp with query, Slide statements
	0	Change value to reference, Combine functions into transform, Move field, Move function, Preserve whole object, Push down method, Remove middle man, Replace command with function, Replace query with parameter

Table 3.2 DeepSeek: Success Rate Per Refactoring Type For Realistic Scenarios (Fowler's Dataset)

LLM	Success Rate	Refactoring Types
DeepSeek	1	Change function declaration, Combine functions into class, Consolidate conditional expression, Decompose conditional, Encapsulate record, Encapsulate variable, Extract superclass, Extract variable, Inline function, Inline variable, Parameterize function, Pull up field, Pull up method, Rename variable, Replace inline code with function call, Replace nested conditional with guard clauses, Replace primitive with object, Replace superclass with delegate, Remove dead code, Replace type code with subclasses, Split loop, Split variable, Substitute algorithm
	0.75	Change value to reference, Collapse hierarchy, Encapsulate collection, Encapsulate variable, Extract class, Extract function, Hide delegate, Introduce assertion, Introduce parameter object, Introduce special case, Inline class, Move field, Move function, Preserve whole object, Push down field, Push down method, Pull up constructor body, Remove flag argument, Remove middle man, Remove subclass, Replace conditional with polymorphism, Replace constructor with factory function, Replace function with command, Replace parameter with query, Replace temp with query, Replace type code with subclasses
	0.67	Combine functions into transform, Move statements into function, Move statements to callers, Rename field, Replace derived variable with query, Replace loop with pipeline, Replace query with parameter, Replace subclass with delegate
	0.5	Change reference to value, Remove setting method, Separate query from modifier, Slide statements
	0.33	Replace command with function, Split phase

Table 3.3 Quality Metrics per Prompt Strategy (Realistic Dataset)

		CodeBLEU	**CC**	**LOC**	**FOUT**
Ground Truth: Before Refactoring			0.443±1.103	3.574±3.243	0.557±1.025
Ground Truth: After Refactoring			0.279±0.773	3.590±3.247	0.836±1.280
GPT-4o Mini	Context Prompt	0.278±0.130	0.459±1.066	11.338±7.262	1.692±2.148
	Instructions Prompt	0.359±0.151	0.479±1.108	14.639±11.098	1.898±2.109
	Rule Prompt	0.359±0.144	0.352±0.867	11.100±7.824	1.243±1.742
	Zero-Shot Prompt	0.303±0.132	0.348±0.838	11.472±8.036	1.518±1.987
DeepSeek	Context Prompt	0.296±0.149	0.311±0.748	9.416±6.763	1.567±2.071
	Instructions Prompt	0.388±0.191	0.489±1.149	13.852±11.655	2.000±2.375
	Rule Prompt	0.407±0.185	0.513±1.026	11.409±9.382	1.400±1.854
	Zero-Shot Prompt	0.359±0.179	0.311±0.827	10.941±9.364	1.607±2.173

Table 3.4 Quality Metrics per Prompt Strategy (Real Dataset)

		CodeBLEU	**CC**	**LOC**	**FOUT**
Ground Truth: Before Refactoring			6.081±5.675	23.735±17.412	10.459±9.279
Ground Truth: After Refactoring			5.820±5.654	23.741±16.586	11.346±10.028
GPT-4o-mini	Context Prompt	0.521±0.052	5.825±5.168	31.359±24.972	12.193±10.749
	Two-Shot Prompt	0.597±0.094	6.120±5.672	26.981±17.112	10.753±9.170
	Instructions Prompt	0.589±0.076	6.008±5.418	27.445±18.715	11.124±9.831
	Rule Prompt	0.597±0.087	6.032±5.624	25.163±15.459	10.599±8.981
	Zero-Shot Prompt	0.558±0.078	5.986±5.446	28.050±18.480	11.455±10.039
DeepSeek	Context Prompt	0.539±0.054	6.052±5.615	27.957±21.200	11.544±10.417
	Two-Shot Prompt	0.534±0.062	6.115±5.609	31.909±19.379	10.854±8.657
	Instructions Prompt	0.530±0.053	6.253±5.576	32.018±18.426	11.217±9.480
	Rule Prompt	0.534±0.059	6.009±5.234	32.144±20.061	11.101±9.325
	Zero-Shot Prompt	0.541±0.060	5.880±5.495	31.803±18.794	11.370±9.092

ground truth supports our argument that LLMs generated method implementations, when the ground truth did not include them.

In our dataset of realistic scenarios, the results show that GPT performs better with *Rule-based* prompts. Specifically, when comparing the GPT prompts, we observe that the *Rule-based* prompt performs statistically significantly better in terms of CodeBLEU compared to the *Context-based* prompt (7th row) [4]. In terms of LOC (5th column), the *Rule-based* prompt (6th row) also performed statistically significantly better than the *Instruction-based* prompt (5th row). In terms of FOUT (6th column), similar results can be observed. Overall, these results show that, on realistic scenarios, GPT performs better with a *Rule-based* prompt, while the *Instruction-based* prompt displays poorer results.

The results for DeepSeek suggest that, for realistic scenarios, this model tends to per-

[4]Mann-Whitney, p-value < 0.05

form more effectively when provided with less context. In fact, when compared with the *Instruction-based* prompt (9th row), the *Context-based* prompt (8th row) exhibited significantly better results with LOC (5th column) and FOUT (6th column)[5]. Similarly, the *Zero-shot* prompt (11th row), one of the prompts that is provided with the least context, showed a significantly better CC (4th column) than the *Rule-based* prompt (10th row), which showed the highest value for this metric. In contrast, just like GPT, the *Instruction-based* prompt yielded the worst performance.

For the dataset of real scenarios, Table 3.4 presents the average metrics collected. The results demonstrate similar values across all prompt types. Similar to the results from realistic scenarios, none of the prompt types achieved a lower CC (4th column) than the human-written code (3rd row).

With GPT, when we compare the prompts with each other, we observe that the *Rule-based* prompt (7th row) and the *Two-shot* (5th row) prompts perform better than the other prompts. In fact, these two prompts show significantly better CodeBLEU score (3rd column) when compared to the *Context-based* prompt (4th row)[6]. The other prompts did not show significant differences between each other for the other three metrics, suggesting that they have a similar impact on code maintainability and readability.

The analysis of the results from DeepSeep suggests that the *Instruction-based* prompt (11th row) performs slightly worse than the other prompts. In fact, this prompt shows the lowest CodeBLEU scores (3rd column), along with some of the highest values for CC (4th column), LOC (5th column) and FOUT (6th column). The statistical analysis shows that the LOC for this prompt is significantly higher than the ground truth before refactoring (3rd row)[7]. With DeepSeek, when we compare the prompts with each other, we observe that the *Zero-shot* prompt (13th row) performs better than the other prompts on real scenarios. This prompt shows significantly better CodeBLEU score (3rd column) when compared to the *Instruction-based* prompt (11th row)[8]. Similar to GPT, the different prompts did not exhibit significant differences in the other metrics, suggesting a similar impact on code maintainability and readability.

[5]Mann-Whitney, p-value < 0.05
[6]Mann-Whitney, p-value < 0.05
[7]Mann-Whitney, p-value < 0.05
[8]Mann-Whitney, p-value < 0.05

3.3 Discussion

In this section, we aim to delve deeper into the implications of our results. Overall, our findings suggest that LLMs demonstrate strong capabilities in performing code refactorings. Notably, the evaluated models successfully handled a broad range of refactoring types, including some that are not even supported by existing assistive tools. As such, LLMs emerge as a promising new tool to assist practitioners in applying refactorings during their daily tasks. Furthermore, it is also important to highlight the complexity of the scenarios under analysis. Although the broader set of refactoring types and more complex scenarios are not real scenarios (realistic dataset), we believe we could show the power of LLMs for such a task, considering the small number of types not successfully addressed in this thesis. On the other hand, it is essential to discuss the impact of applied refactorings on code quality. Overall, we observe that LLMs encounter challenges in performing refactorings while maintaining or enhancing code quality. Even when a refactoring is successfully applied, the modifications introduced by LLMs can have a direct impact on various quality aspects, as demonstrated by the metrics we computed (CC, LOC, FOUT).

Regarding the LLMs under analysis, GPT and DeepSeek exhibit similar strengths and challenges. For instance, both models struggle to apply refactorings while simultaneously improving the quality of the code. When evaluating our sample of real scenarios, some refactorings resulted in more problematic solutions—such as non-compilable code or failing tests. Given that preserving behavior is a core principle of refactoring, these challenges highlight the need for further improvements in LLM-generated refactorings, particularly in ensuring both syntactic and semantic correctness.

In this thesis, we employed five different prompts to evaluate the LLMs, aiming to maximize their effectiveness through diverse strategies. While all prompts led to similar conclusions regarding the quality of the generated code, we found that the *Rule-based* prompt yielded better results with GPT-4o mini, whereas DeepSeek performed better when given less context.

The key distinction of the *Rule-based* prompt lies in its provision of more precise information, clearly defining the expected changes, while the other prompts offered broader or less constrained instructions.

3.4 Chapter Summary

This chapter provides an overview of this thesis.

Section 3.1 details the methodology employed to evaluate the effectiveness of LLMs in code

refactoring tasks. It describes the steps taken to collect datasets of realistic and real-world refactorings, define various prompt strategies, and select LLMs for evaluation. The process begins with the collection of refactoring scenarios from two sources: Fowler's Catalog of Refactorings and the Ref-Finder tool dataset. For each dataset, refactorings are manually validated to ensure their accuracy.

Next, the methodology focuses on the design of different prompting strategies for interacting with LLMs, including *Zero-shot*, *Two-shot*, *Instruction-based*, *Rule-based*, and *Context-based* prompts. Each prompt is intended to evaluate how LLMs respond to varying levels of input information. The LLMs selected for this thesis are GPT-4o-mini and DeepSeek-V3, both state-of-the-art models designed for natural language processing tasks.

The thesis proceeds with the prompting of these models using carefully curated scenarios, followed by a manual validation process to assess the correctness of the refactorings in the case of realistic scenarios. Additionally, an automatic validation process is employed for real-world refactorings, where code is tested for compilation success and semantic accuracy.

Furthermore, the methodology includes the calculation of several metrics, such as CodeBLEU and cyclomatic complexity, to evaluate the quality of the LLM-generated code.

Section 3.2 and Section 3.3 present the findings of the thesis. The results indicate that GPT-4o-mini performs most effectively when presented with a *Rule-based* prompt. Additionally, DeepSeek demonstrated better performance with limited contextual information, as both *Context-based* and *Zero-shot* prompts produced higher success rates.

CHAPTER 4 CONCLUSION

In this chapter, we summarize the work presented in this thesis, discuss the limitations of our experiments, and suggest potential avenues for future research.

4.1 Summary of Works

The rapid advancement of LLMs has sparked significant interest in their potential applications in software engineering tasks, particularly in the area of code generation. Previous research has delved into the use of LLMs for code refactoring, but most studies have focused on relatively simple refactoring tasks, leaving a substantial gap in exploring the full range of capabilities these models might offer. This study aims to address this gap by investigating the ability of LLMs to perform a wider variety of refactoring types, while also examining the impact of different prompting strategies on code quality.

To conduct this study, we utilize two distinct datasets: one derived from Martin Fowler's well-known refactoring catalog (realistic scenarios), and another sourced from real-world GitHub projects, offering a diverse range of practical, community-driven code examples. The study is carried out using two state-of-the-art LLMs: OpenAI's GPT-4o-mini and DeepSeek-V3. These models were selected to assess their performances across various quality metrics, as well as their ability to maintain the semantic integrity of the code during refactoring.

Our findings suggest that LLMs are highly capable of performing code refactorings, even for more complex and intricate scenarios. This thesis marks an important first step toward addressing more advanced refactoring tasks that are not yet supported by existing assistive tools. Notably, our results show that DeepSeek outperformed GPT in handling a wider range of refactoring types. However, both models encountered challenges in applying some refactoring types without compromising the quality metrics of the code.

In terms of prompting strategies, our analysis revealed that GPT performed optimally when provided with *Rules-based* or *Two-shot* prompts, yielding higher CodeBLEU scores. This suggests that these prompts lead to a closer resemblance to human-written code. In contrast, DeepSeek achieved higher scores when prompted with a *Zero-shot* prompt, indicating that it performs better with less contextual information.

4.2 Limitations

This thesis is subject to several potential threats that could influence the results presented here. In the following section, we identify these threats and discuss the strategies we employed to mitigate their impact.

Construct to Validity. To evaluate the correctness of the refactorings performed by LLMs on realistic scenarios, we conducted a manual analysis. To minimize potential bias, two researchers independently assessed the refactorings and then discussed any disagreements to reach a consensus. As previously mentioned, we observed high Cohen's Kappa coefficients, indicating strong agreement between the reviewers. Additionally, we explored the use of LLMs as judges, as suggested by Zhao et al. [59]. However, when compared to human judgment, we found that LLMs tend to be more conservative, often demonstrating a greater tendency to accept proposed refactorings as correct.

To assess the semantic impact of the generated refactorings, we relied on running the test suite associated with the target project. However, if the project's test suite is inadequate, it may fail to detect regressions, potentially leading to inaccurate conclusions regarding the correctness of the refactorings. To mitigate this risk, we selected popular projects known for having robust test suites. Additionally, we acknowledged the potential for flaky tests to introduce variability in the test results. To address this concern, we executed the tests five times for each scenario and observed no instances of test flakiness.

In designing our *Rule-based* prompt, we did not establish rules for all the refactoring types under analysis. As a result, 15 refactoring types were not included in the prompt, potentially limiting the LLM's ability to generate refactorings effectively. This limitation may introduce bias into our assessment, as the study might not fully capture the LLM's performance across a comprehensive set of refactoring tasks using this prompt. Furthermore, refactoring types that require more structured guidance may be underrepresented, which could influence the overall conclusions of the study.

Another potential threat arises from our use of CodeBLEU to evaluate the refactored code. During our analysis, we observed that LLMs received lower CodeBLEU scores not due to failures in applying the requested refactoring, but because they generated additional code that was not strictly necessary. While the model correctly performed the refactoring as specified in the prompt in some cases, it also included portions of the original code, leading to discrepancies when compared to the ground truth. This limitation suggests that CodeBLEU may not fully capture the correctness of refactoring in such instances, potentially underestimating the true performance of LLMs in performing the refactorings.

Internal Validity. Our results may be subject to bias due to the non-deterministic behavior of LLMs. To mitigate this, and in line with previous studies, we prompted each LLM multiple times (five) for each refactoring scenario to reduce the impact of this variability. Another potential threat to internal validity is the presence of biases in our dataset arising from memorization, as discussed by Carlini et al. [60]. Given that we selected realistic and real-world refactoring scenarios, there is a possibility that these scenarios were encountered during the training of the LLMs, which could influence the results.

External to Validity. Our findings are limited to the context of a single programming language, Java. While Java is widely used, restricting the study to this language may result in biased conclusions, as different programming languages have unique syntax, rules, and assistive tools. Similarly, we evaluated the capabilities of two LLMs, both of which are commonly used for various tasks in software engineering. Conducting this study with other models, or employing advanced adaptation techniques such as fine-tuning, could yield different results.

4.3 Future Research

While this study provides initial insights into the potential of LLMs for code refactoring, several limitations highlight important avenues for future research.

Expanding the evaluation to encompass a wider range of programming languages would deepen our understanding of how LLMs perform across different syntactic structures and language-specific rules. Building on our findings, future studies should further investigate the ability of LLMs to perform refactorings, with a particular focus on improving code quality.

An additional important avenue for research is the exploration of a wider variety of LLMs, as well as the integration of advanced adaptation techniques like fine-tuning, to determine whether these approaches can lead to improved performance.

Moreover, future work could explore the development of autonomous systems capable of performing continuous refactoring in real-time as developers work on their codebases. These agents, which could be integrated directly into IDEs, could autonomously detect code smells and refactor them without the need for explicit prompts or user interventions.

Finally, future research could explore the benefits of integrating LLMs with human oversight in the refactoring process. This "human-in-the-loop" approach would help ensure that the refactored code maintains its intended meaning and functionality, addressing any issues related to semantic accuracy. Additionally, involving humans could improve the transparency of the model's decision-making, making it easier to understand how the LLM arrives at its conclusions and ensuring the quality of the refactorings in real-world applications.

REFERENCES

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

[2] A. Shirafuji *et al.*, "Refactoring programs using large language models with few-shot examples," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2023, p. 151–160. [Online]. Available: http://dx.doi.org/10.1109/APSEC60848.2023.00025

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2018.

[4] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[5] C. S. Tavares, F. Ferreira, and E. Figueiredo, "A systematic mapping of literature on software refactoring tools," in *Proceedings of the XIV Brazilian Symposium on Information Systems*, ser. SBSI '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3229345.3229357

[6] A. Brito, A. Hora, and M. T. Valente, "Refactoring graphs: Assessing refactoring over time," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 367–377.

[7] M. M. Rahman *et al.*, "An empirical study on the occurrences of code smells in open source and industrial projects," in *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–294. [Online]. Available: https://doi.org/10.1145/3544902.3546634

[8] F. Niu *et al.*, "Rat: A refactoring-aware tool for tracking code history," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 104–108. [Online]. Available: https://doi.org/10.1145/3639478.3640047

[9] M. T. Hasan, N. Tsantalis, and P. Alikhanifard, "Refactoring-aware block tracking in commit history," 2024. [Online]. Available: https://arxiv.org/abs/2409.16185

[10] Z. Wang *et al.*, "History, development, and principles of large language models-an introductory survey," 2024. [Online]. Available: https://arxiv.org/abs/2402.06853

[11] G. Bavota *et al.*, "Supporting extract class refactoring in eclipse: The aries project," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 1419–1422.

[12] D. Silva, R. Terra, and M. T. Valente, "Jextract: An eclipse plug-in for recommending automated extract method refactorings," 2015. [Online]. Available: https://arxiv.org/abs/1506.06086

[13] R. M. Fuhrer, A. Kiezun, and M. Keller, "Refactoring in the eclipse jdt : Past , present , and future," 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:15621533

[14] A. M. Eilertsen and G. C. Murphy, "The usability (or not) of refactoring tools," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 237–248.

[15] N. Tsantalis *et al.*, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 483–494. [Online]. Available: https://doi.org/10.1145/3180155.3180206

[16] M. Kim *et al.*, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 371–372. [Online]. Available: https://doi.org/10.1145/1882291.1882353

[17] H. Liu *et al.*, "Refbert: A two-stage pre-trained framework for automatic rename refactoring," 2023. [Online]. Available: https://arxiv.org/abs/2305.17708

[18] D. Pomian *et al.*, "Together we go further: Llms and ide static analysis for extract method refactoring," 2024. [Online]. Available: https://arxiv.org/abs/2401.15298

[19] J. Choi, G. An, and S. Yoo, "Iterative refactoring of real-world open-source programs with large language models," in *Search-Based Software Engineering*, G. Jahangirova and F. Khomh, Eds. Cham: Springer Nature Switzerland, 2024, pp. 49–55.

[20] J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2411.02320

[21] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2018.

[22] D. Das *et al.*, "Comex: A tool for generating customized source code representations," 2023. [Online]. Available: https://arxiv.org/abs/2307.04693

[23] S. Khanzadeh *et al.*, "Opti code pro: A heuristic search-based approach to code refactoring," 2023. [Online]. Available: https://arxiv.org/abs/2305.07594

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

[25] JetBrains, "Intellij idea," 2025. [Online]. Available: https://www.jetbrains.com/idea/

[26] E. Foundation, "Eclipse ide," 2025. [Online]. Available: https://www.eclipse.org/

[27] Microsoft, "Visual studio code," 2025. [Online]. Available: https://code.visualstudio.com/

[28] J. Team, "Jrefactory," 2005. [Online]. Available: https://jrefactory.sourceforge.net/

[29] E. Team, "Eclim," 2020. [Online]. Available: https://eclim.org/

[30] V. S. C. Team, "Visual studio code: Refactoring," 2025. [Online]. Available: https://code.visualstudio.com/docs/editing/refactoring

[31] JetBrains, "Intellij idea refactoring support," 2025. [Online]. Available: https://www.jetbrains.com/help/idea/refactoring-source-code.html

[32] N. Wadhwa *et al.*, "Core: Resolving code quality issues using llms," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643762

[33] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," 2024. [Online]. Available: https://arxiv.org/abs/2403.17134

[34] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[35] D. de Fitero-Dominguez *et al.*, "Enhanced automated code vulnerability repair using large language models," *Engineering Applications of Artificial Intelligence*, vol. 138, p. 109291, Dec. 2024. [Online]. Available: http://dx.doi.org/10.1016/j.engappai.2024.109291

[36] I. R. da Silva Simões and E. Venson, "Evaluating source code quality with large language models: a comparative study," 2024. [Online]. Available: https://arxiv.org/abs/2408.07082

[37] Y. Wang *et al.*, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[38] M. Chen *et al.*, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[39] E. Nijkamp *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," 2023. [Online]. Available: https://arxiv.org/abs/2203.13474

[40] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," 2025. [Online]. Available: https://arxiv.org/abs/2401.11817

[41] J. Devlin *et al.*, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[42] OpenAI, "Gpt-4o mini: advancing cost-efficient intelligence," 2024. [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[43] J. Jiang *et al.*, "A survey on large language models for code generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[44] A. Anand *et al.*, "A comprehensive survey of ai-driven advancements and techniques in automated program repair and code generation," 2024. [Online]. Available: https://arxiv.org/abs/2411.07586

[45] I. Guelman *et al.*, "Using large language models to document code: A first quantitative and qualitative assessment," 2024. [Online]. Available: https://arxiv.org/abs/2408.14007

[46] A. Fan *et al.*, "Large language models for software engineering: Survey and open problems," 2023. [Online]. Available: https://arxiv.org/abs/2310.03533

[47] M. Fowler, "Catalog of refactorings." [Online]. Available: https://refactoring.com/catalog/

[48] I. Kádár *et al.*, "A manually validated code refactoring dataset and its assessment regarding software maintainability," in *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2016. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2972958.2972962

[49] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: https://doi.org/10.1177/001316446002000104

[50] W. Wang *et al.*, "A survey of zero-shot learning: Settings, methods, and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3293318

[51] Y. Wang *et al.*, "Generalizing from a few examples: A survey on few-shot learning," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3386252

[52] K. Prete *et al.*, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[53] K. Prete, N. Rachatasumrit, and M. Kim, "Catalogue of template refactoring rules," *The University of Texas at Austin, Tech. Rep. UTAUSTINECE-TR-041610*, 2010.

[54] D. Guo *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[55] S. Ren *et al.*, "Codebleu: a method for automatic evaluation of code synthesis," 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[56] K. Papineni *et al.*, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA: Association for Computational Linguistics, 2002, p. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[57] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[58] M. M. Mohajer, "pyccmetrics," https://github.com/mmohajer9/pyccmetrics, 2022.

[59] Y. Zhao *et al.*, "Codejudge-eval: Can large language models be good judges in code understanding?" *arXiv preprint arXiv:2408.10718*, 2024.

[60] N. Carlini *et al.*, "Quantifying memorization across neural language models," in *The Eleventh International Conference on Learning Representations*, 2022.

APPENDIX A QUALITY METRICS FOR REALISTIC SCENARIOS (FOWLER'S DATASET) AND REAL SCENARIOS

This section includes the tables that present the quality metrics for realistic scenarios, the compilation data and semantic analysis of the real scenarios, and the quality metrics of real scenarios mentioned in Section 3.2

Table A.1 GPT: Quality Metrics For Realistic Scenarios
(Fowler's Dataset)

Refactoring Type	**CodeBleu**	**CC**	**LOC**	**FOUT**
Change Function Declaration	0.262 ± 0.024	0.000 ± 0.000	4.900 ± 3.173	0.000 ± 0.000
Change Reference To Value	0.236 ± 0.057	0.500 ± 1.000	18.900 ± 10.390	1.750 ± 1.075
Change Value To Reference	0.208 ± 0.044	0.000 ± 0.000	8.150 ± 12.300	1.250 ± 1.491
Collapse Hierarchy	0.321 ± 0.006	0.000 ± 0.000	5.650 ± 4.300	0.100 ± 0.200
Combine Functions Into Class	0.322 ± 0.022	0.267 ± 0.306	19.800 ± 1.058	3.400 ± 0.872
Combine Functions Into Transform	0.146 ± 0.039	0.200 ± 0.200	18.000 ± 7.186	2.667 ± 0.945
Consolidate Condition Expression	0.191 ± 0.001	2.000 ± 0.000	4.750 ± 2.323	0.250 ± 0.500
Decompose Conditional	0.148 ± 0.004	1.400 ± 0.542	10.800 ± 4.907	3.750 ± 1.269
Encapsulate Collection	0.307 ± 0.002	0.000 ± 0.000	13.700 ± 3.447	1.500 ± 1.732
Encapsulate Record	0.213 ± 0.028	0.000 ± 0.000	20.100 ± 4.957	1.050 ± 1.843
Encapsulate Variable	0.140 ± 0.015	0.000 ± 0.000	19.400 ± 3.274	0.150 ± 0.300
Encapsulate Class	0.326 ± 0.042	0.000 ± 0.000	23.300 ± 6.133	1.300 ± 0.887
Extract Function	0.426 ± 0.200	0.000 ± 0.000	12.700 ± 3.866	5.650 ± 0.915
Extract Superclass	0.336 ± 0.022	0.000 ± 0.000	24.850 ± 8.487	0.000 ± 0.000
Extract Variable	0.568 ± 0.121	0.150 ± 0.300	6.850 ± 2.955	2.050 ± 0.100
Hide Delegate	0.170 ± 0.063	0.000 ± 0.000	10.400 ± 12.891	1.750 ± 0.790
Inline Class	0.371 ± 0.038	0.000 ± 0.000	19.450 ± 5.149	1.200 ± 0.924
Inline Function	0.486 ± 0.085	1.000 ± 0.000	6.250 ± 4.500	0.250 ± 0.500
Inline Variable	0.294 ± 0.011	0.000 ± 0.000	2.400 ± 0.800	0.250 ± 0.500
Introduce Assertion	0.439 ± 0.225	0.250 ± 0.500	4.250 ± 0.300	0.050 ± 0.100
Introduce Parameter Object	0.279 ± 0.021	0.000 ± 0.000	22.150 ± 8.822	2.700 ± 2.049

Introduce Special Case	0.266 ± 0.016	0.350 ± 0.473	10.000 ± 11.520	1.650 ± 1.455
Move Field	0.341 ± 0.023	0.000 ± 0.000	12.150 ± 1.921	0.000 ± 0.000
Move Function	0.351 ± 0.093	0.100 ± 0.200	8.300 ± 3.260	0.250 ± 0.300
Move Statements Into Function	0.410 ± 0.200	0.000 ± 0.000	12.467 ± 0.987	6.600 ± 3.124
Move Statements To Callers	0.194 ± 0.057	0.000 ± 0.000	4.133 ± 1.963	2.400 ± 3.292
Parametrize Function	0.310 ± 0.144	0.000 ± 0.000	11.100 ± 1.943	3.750 ± 0.870
Preserve Whole Object	0.297 ± 0.021	0.650 ± 0.300	7.700 ± 4.058	4.050 ± 1.340
Pull Up Constructor Body	0.435 ± 0.045	0.000 ± 0.000	18.800 ± 6.997	0.000 ± 0.000
Pull Up Field	0.567 ± 0.076	0.000 ± 0.000	12.500 ± 5.219	0.000 ± 0.000
Pull Up Method	0.452 ± 0.002	0.000 ± 0.000	13.450 ± 1.955	0.100 ± 0.200
Push Down Field	0.547 ± 0.033	0.000 ± 0.000	10.250 ± 2.500	0.000 ± 0.000
Push Down Method	0.392 ± 0.087	0.000 ± 0.000	11.200 ± 2.668	0.000 ± 0.000
Remove Dead Code	0.250 ± 0.000	0.000 ± 0.000	1.000 ± 0.000	0.000 ± 0.000
Remove Flag Argument	0.316 ± 0.050	0.750 ± 1.500	8.700 ± 2.891	0.050 ± 0.100
Remove Middle Man	0.156 ± 0.007	0.000 ± 0.000	10.950 ± 0.985	0.950 ± 0.100
Remove Setting Method	0.350 ± 0.000	0.000 ± 0.000	10.000 ± 0.000	0.000 ± 0.000
Remove Subclass	0.344 ± 0.016	0.000 ± 0.000	17.350 ± 1.843	0.300 ± 0.600
Rename Field	0.345 ± 0.002	0.000 ± 0.000	8.200 ± 1.587	0.000 ± 0.000
Rename Variable	0.737 ± 0.041	0.000 ± 0.000	2.000 ± 0.000	0.000 ± 0.000
Replace Command With Function	0.304 ± 0.004	0.000 ± 0.000	4.400 ± 3.050	0.500 ± 0.100
Replace Conditional With Polymorphism	0.299 ± 0.027	0.500 ± 1.000	7.400 ± 2.059	1.000 ± 1.732
Replace Constructor With Factory Function	0.284 ± 0.021	0.000 ± 0.000	18.350 ± 8.944	1.200 ± 1.600
Replace Derived Variable With Query	0.227 ± 0.009	0.000 ± 0.000	7.750 ± 2.127	0.200 ± 0.400
Replace Function With Command	0.190 ± 0.085	0.000 ± 0.000	14.200 ± 6.409	0.250 ± 0.500
Replace Inline Code With Function Call	0.225 ± 0.081	0.000 ± 0.000	7.000 ± 4.207	0.500 ± 0.500

Replace Loop With Pipeline	0.206 ± 0.211	0.000 ± 0.000	8.600 ± 4.234	0.500 ± 0.500
Replace Nested Conditional With Guard Clauses	0.638 ± 0.108	0.000 ± 0.000	12.200 ± 7.843	0.000 ± 0.000
Replace Parameter With Query	0.241 ± 0.061	0.000 ± 0.000	3.800 ± 1.868	0.000 ± 0.000
Replace Primitive With Object	0.260 ± 0.010	0.000 ± 0.000	10.300 ± 3.547	1.250 ± 0.250
Replace Query With Parameter	0.319 ± 0.081	0.000 ± 0.000	8.100 ± 2.000	0.000 ± 0.000
Replace Subclass With Delegate	0.298 ± 0.074	0.000 ± 0.000	15.800 ± 4.236	0.250 ± 0.250
Replace Superclass With Delegate	0.250 ± 0.079	0.000 ± 0.000	12.500 ± 2.193	0.250 ± 0.500
Replace Temp With Query	0.239 ± 0.103	0.000 ± 0.000	7.600 ± 4.428	0.250 ± 0.250
Replace Type Code With Subclasses	0.322 ± 0.087	0.000 ± 0.000	14.750 ± 5.764	0.000 ± 0.000
Separate Query From Modifier	0.302 ± 0.064	0.000 ± 0.000	6.500 ± 4.115	0.000 ± 0.000
Slide Statements	0.450 ± 0.110	0.000 ± 0.000	10.250 ± 5.831	1.600 ± 0.548
Split Loop	0.326 ± 0.106	0.000 ± 0.000	9.600 ± 2.758	0.000 ± 0.000
Split Phase	0.155 ± 0.041	0.000 ± 0.000	5.400 ± 4.850	0.000 ± 0.000
Split Variable	0.616 ± 0.026	0.000 ± 0.000	7.250 ± 2.315	0.000 ± 0.000
Substitute Algorithm	0.183 ± 0.024	0.000 ± 0.000	9.500 ± 3.107	0.000 ± 0.000

Table A.2 DeepSeek: Quality Metrics For Realistic Scenarios (Fowler's Dataset)

Refactoring Type	**CodeBleu**	**CC**	**LOC**	**FOUT**
Change Function Declaration	0.251 ± 0.003	0.200 ± 0.400	4.250 ± 1.893	0.200 ± 0.400
Change Reference To Value	0.298 ± 0.012	1.800 ± 2.135	26.750 ± 10.180	4.150 ± 2.563
Change Value To Reference	0.221 ± 0.065	0.050 ± 0.100	6.300 ± 8.600	1.950 ± 2.068
Collapse Hierarchy	0.332 ± 0.023	0.000 ± 0.000	9.750 ± 10.882	0.200 ± 0.400
Combine Functions Into Class	0.346 ± 0.027	0.000 ± 0.000	21.467 ± 9.047	3.067 ± 0.503
Combine Functions Into Transform	0.214 ± 0.069	0.000 ± 0.000	25.533 ± 6.313	4.467 ± 3.585
Consolidate Conditional Expression	0.191 ± 0.001	2.000 ± 0.000	5.300 ± 1.536	0.400 ± 0.490
Decompose Conditional	0.150 ± 0.012	1.400 ± 0.490	10.600 ± 5.906	3.750 ± 1.500
Encapsulate Collection	0.308 ± 0.005	0.000 ± 0.000	13.950 ± 3.743	2.600 ± 1.083
Encapsulate Record	0.221 ± 0.029	0.000 ± 0.000	23.750 ± 2.568	1.050 ± 1.969
Encapsulate Variable	0.176 ± 0.040	0.000 ± 0.000	25.450 ± 5.660	0.450 ± 0.526
Extract Class	0.299 ± 0.049	0.000 ± 0.000	21.000 ± 6.765	1.900 ± 1.332
Extract Function	0.228 ± 0.012	0.000 ± 0.000	13.550 ± 5.446	5.800 ± 0.400
Extract Superclass	0.675 ± 0.173	0.000 ± 0.000	19.700 ± 6.622	0.700 ± 0.476
Extract Variable	0.673 ± 0.019	0.000 ± 0.000	5.000 ± 0.000	2.000 ± 0.000
Hide Delegate	0.139 ± 0.035	0.000 ± 0.000	3.350 ± 2.700	1.500 ± 0.577
Inline Class	0.508 ± 0.145	0.000 ± 0.000	11.750 ± 6.777	0.500 ± 1.000
Inline Function	0.453 ± 0.025	1.000 ± 0.000	4.000 ± 0.000	0.250 ± 0.500
Inline Variable	0.405 ± 0.232	0.000 ± 0.000	2.000 ± 0.000	0.000 ± 0.000
Introduce Assertion	0.532 ± 0.220	1.000 ± 0.816	4.000 ± 0.816	0.000 ± 0.000
Introduce Parameter Object	0.285 ± 0.023	0.000 ± 0.000	19.400 ± 7.200	0.000 ± 0.000

Introduce Special Case	0.286 ± 0.016	0.300 ± 0.476	15.600 ± 10.019	1.750 ± 0.900
Move Field	0.381 ± 0.042	0.000 ± 0.000	16.400 ± 6.487	1.050 ± 0.823
Move Function	0.413 ± 0.189	0.050 ± 0.100	7.500 ± 3.109	0.400 ± 0.800
Move Statements Into Function	0.425 ± 0.321	0.000 ± 0.000	9.733 ± 1.270	6.533 ± 4.406
Move Statements To Callers	0.186 ± 0.029	0.000 ± 0.000	4.000 ± 1.732	3.600 ± 4.678
Parametrize Function	0.268 ± 0.092	0.000 ± 0.000	4.000 ± 0.000	2.600 ± 0.993
Preserve Whole Object	0.363 ± 0.018	0.200 ± 0.400	2.700 ± 1.400	1.200 ± 0.400
Pull Up Constructor Body	0.707 ± 0.215	0.000 ± 0.000	13.650 ± 1.300	0.750 ± 0.500
Pull Up Field	0.588 ± 0.066	0.000 ± 0.000	11.750 ± 7.500	0.000 ± 0.000
Pull Up Method	0.512 ± 0.095	0.000 ± 0.000	11.400 ± 5.748	0.400 ± 0.490
Push Down Field	0.542 ± 0.029	0.000 ± 0.000	9.800 ± 2.926	0.000 ± 0.000
Push Down Method	0.550 ± 0.075	0.000 ± 0.000	10.350 ± 1.799	0.000 ± 0.000
Remove Dead Code	0.250 ± 0.000	0.000 ± 0.000	1.000 ± 0.000	0.000 ± 0.000
Remove Flag Argument	0.336 ± 0.056	0.550 ± 1.100	7.650 ± 1.300	0.300 ± 0.600
Remove Middle Man	0.256 ± 0.070	0.000 ± 0.000	10.000 ± 4.690	1.700 ± 0.600
Remove Setting Method	0.351 ± 0.003	0.000 ± 0.000	9.400 ± 1.200	0.000 ± 0.000
Remove Subclass	0.351 ± 0.016	0.000 ± 0.000	16.350 ± 3.869	0.050 ± 0.100
Rename Field	0.470 ± 0.214	0.000 ± 0.000	7.000 ± 3.000	0.000 ± 0.000
Rename Variable	0.708 ± 0.024	0.000 ± 0.000	2.000 ± 0.000	0.000 ± 0.000
Replace Command With Function	0.410 ± 0.186	0.000 ± 0.000	10.000 ± 3.464	0.667 ± 0.577
Replace Conditional With Polymorphism	0.294 ± 0.043	3.000 ± 2.000	33.500 ± 16.583	0.150 ± 0.300
Replace Constructor With Factory Function	0.287 ± 0.035	0.000 ± 0.000	3.500 ± 1.428	1.000 ± 0.000
Replace Derived Variable With Query	0.308 ± 0.021	0.000 ± 0.000	9.867 ± 0.808	0.733 ± 0.643
Replace Function With Command	0.259 ± 0.127	0.000 ± 0.000	16.450 ± 7.473	0.400 ± 0.490
Replace Inline Code With Function Call	0.182 ± 0.000	0.000 ± 0.000	2.000 ± 0.000	1.000 ± 0.000

Replace Loop With Pipeline	0.035 ± 0.003	0.333 ± 0.577	5.667 ± 1.155	6.667 ± 2.309
Replace Nested Conditional With Guard Clauses	0.720 ± 0.196	3.000 ± 0.000	10.300 ± 3.156	4.000 ± 0.000
Replace Parameter With Query	0.430 ± 0.172	0.000 ± 0.000	4.400 ± 0.800	0.500 ± 0.872
Replace Primitive With Object	0.269 ± 0.008	0.900 ± 0.115	7.700 ± 7.812	6.300 ± 1.732
Replace Query With Parameter	0.374 ± 0.209	0.000 ± 0.000	3.667 ± 0.577	0.667 ± 0.577
Replace Subclass With Delegate	0.315 ± 0.093	0.800 ± 0.721	25.933 ± 4.388	2.333 ± 0.577
Replace Superclass With Delegate	0.322 ± 0.065	0.050 ± 0.100	15.100 ± 7.006	2.000 ± 4.000
Replace Temp With Query	0.342 ± 0.121	0.750 ± 0.500	8.550 ± 2.247	2.250 ± 1.500
Replace Type Code With Subclasses	0.387 ± 0.117	2.750 ± 1.893	36.750 ± 22.846	0.000 ± 0.000
Separate Query From Modifier	0.271 ± 0.054	0.000 ± 0.000	10.800 ± 0.993	5.350 ± 0.473
Slide Statements	0.492 ± 0.165	0.000 ± 0.000	4.950 ± 0.100	2.500 ± 0.577
Split Loop	0.228 ± 0.116	0.000 ± 0.000	7.667 ± 4.041	2.933 ± 3.635
Split Phase	0.157 ± 0.054	0.000 ± 0.000	11.800 ± 10.440	4.600 ± 1.217
Split Variable	0.632 ± 0.003	0.000 ± 0.000	5.000 ± 0.000	2.000 ± 0.000
Substitute Algorithm	0.180 ± 0.000	3.000 ± 0.000	9.000 ± 0.000	3.000 ± 0.000

Table A.3 Compilation Data and Semantic Analysis Per Refactoring Type (Real Dataset)

LLM	Refactoring Type	Avg. Compilation	Avg. New Failed Tests	Avg. New Test Errors
GPT-4o m.	Split Variable	0.967±0.082	0	0
	Extract Variable	0.910±0.279	0	0
	Extract Function	0.611±0.473	0	0
	Replace NC with Guard Clauses	0.506±0.480	3.388±8.861	1.412±3.692
	Replace Function with Command	0.480±0.510	0	0
	Consolidate Cond. Expression	0.416±0.493	0.008±0.040	0.016±0.080
	Change Function Declaration	0.352±0.452	0	0
	Introduce Assertion	0.200±0.400	0	0
	Slide Statements	0.183±0.371	0.026±0.113	0
	Inline Variable	0	0	0
	Introduce Special Case	0	0	0
DeepSeek	Split Variable	1	0	0
	Extract Variable	0.940±0.226	0	0
	Extract Function	0.676±0.475	0	0
	Replace Function with Command	0.407±0.494	0	0
	Consolidate Cond. Expression	0.400±0.500	0	0
	Replace NC with Guard Clauses	0.307±0.453	0.033±0.129	0
	Change Function Declaration	0.224±0.410	0.020±0.100	0
	Slide Statements	0.155±0.360	0.045±0.251	0
	Introduce Special Case	0	0	0
	Inline Variable	0	0	0
	Introduce Assertion	0	0	0

Table A.4 Quality Metrics per Refactoring Types (Real Scenarios)

LLM	Refactoring Type	CodeBleu	CC	LOC	FOUT
GPT-4o m.	Introduce Assertion	0.667±0.065	0.440±0.121	13.493±1.175	7.227±0.358
	Introduce Special Case	0.658±0.043	17.320±0.743	67.480±12.032	26.040±3.160
	Split Variable	0.640±0.048	1.500±0	12.280±4.160	1.360±0.699
	Extract Function	0.596±0.044	3.937±0.175	23.634±1.564	10.171±0.326
	Replace NC With Guard Clauses	0.595±0.042	12.480±0.444	46.620±5.330	15.580±1.411
	Slide Statements	0.592±0.106	10.144±0.062	48.888±6.687	31.628±1.508
	Consolidate Cond. Expression	0.557±0.050	7.648±0.212	27.528±2.854	12.040±0.616
	Replace Function With Command	0.536±0.022	6.187±0.198	24.267±6.853	2.860±0.155
	Change Function Declaration	0.518±0.019	4.280±0.073	20.475±1.941	6.956±0.447
	Inline Variable	0.496±0.023	1.000±0	9.440±0.984	3.080±0.179
	Extract Variable	0.442±0.021	1.000±0	11.690±1.118	6.530±0.347
DeepSeek	Split Variable	0.620±0.021	1.500±0	12.080±2.027	1.240±0.428
	Introduce Special Case	0.582±0.017	16.800±1.049	69.600±4.626	24.600±2.025
	Replace NC With Guard Clauses	0.580±0.019	13.420±0.559	52.220±0.998	16.100±0.686
	Introduce Assertion	0.579±0.031	0.480±0.159	16.693±2.547	7.453±0.568
	Slide Statements	0.549±0.030	10.064±0.288	49.708±2.479	30.828±1.819
	Extract Function	0.547±0.021	4.086±0.169	28.423±4.748	10.531±0.866
	Consolidate Conditional Expression	0.516±0.022	8.296±0.582	33.400±2.860	13.032±0.648
	Replace Function With Command	0.498±0.028	5.880±0.939	33.913±11.643	3.167±1.056
	Inline Variable	0.497±0.020	1.000±0	10.080±0.782	3.000±0
	Change Function Declaration	0.479±0.029	4.151±0.166	24.040±2.935	6.849±0.338
	Extract Variable	0.443±0.011	1.000±0	12.670±1.281	6.590±0.216

APPENDIX B CO-AUTHORSHIP

The research study in this thesis was submitted as follows:

- Refactoring with LLMs: Insights from Martin Fowler's Catalog, **Yonnel Chen Kuang Piao**, Jean Carlors Paul, Leuson Da Silva, Arghavan Moradi Dakhel, Mohammad Hamdaqa and Foutse Khomh to *41st IEEE International Conference on Software Maintenance and Evolution (ICSME 2025)*