RefModel: Detecting Refactorings using Foundation Models

Pedro Simões
Federal Univ. of Campina Grande
Campina Grande, PB, Brazil
pedro.henrique.lima.simoes@ccc.ufcg.edu.br

Rohit Gheyi
Federal Univ. of Campina Grande
Campina Grande, PB, Brazil
rohit@dsc.ufcg.edu.br

Rian Melo
Federal Univ. of Campina Grande
Campina Grande, PB, Brazil
rian.melo@ccc.ufcg.edu.br

Jonhnanthan Oliveira
Federal Univ. of Campina Grande
Campina Grande, PB, Brazil
jonhnanthan@copin.ufcg.edu.br

Márcio Ribeiro
Federal University of Alagoas
Maceió, AL, Brazil
marcio@ic.ufal.br

Wesley K. G. Assunção
North Carolina State University
Raleigh, NC, United States
wguezas@ncsu.edu

ABSTRACT

Refactoring is a common software engineering practice that improves code quality without altering program behavior. Although tools like ReExtractor+, RefactoringMiner, and RefDiff have been developed to detect refactorings automatically, they rely on complex rule definitions and static analysis, making them difficult to extend and generalize to other programming languages. In this paper, we investigate the viability of using foundation models for refactoring detection, implemented in a tool named RefModel. We evaluate Phi4-14B, and Claude 3.5 Sonnet on a dataset of 858 single-operation transformations applied to artificially generated Java programs, covering widely-used refactoring types. We also extend our evaluation by including Gemini 2.5 Pro and o4-mini-high, assessing their performance on 44 real-world refactorings extracted from four open-source projects. These models are compared against RefactoringMiner, RefDiff, and ReExtractor+. RefModel is competitive with, and in some cases outperform, traditional tools. In real-world settings, Claude 3.5 Sonnet and Gemini 2.5 Pro jointly identified 97% of all refactorings, surpassing the best-performing static-analysis-based tools. The models showed encouraging generalization to Python and Golang. They provide natural language explanations and require only a single sentence to define each refactoring type.

KEYWORDS

Refactoring Detection, Foundation Models, RefModel

1 Introduction

Refactoring is the process of modifying a program's internal structure to improve readability, maintainability, and design quality, while preserving its external behavior [14, 25, 35, 36]. Recent studies show that more than 40% of developers perform refactorings on a daily basis [17]. Accurate refactoring detection is essential for understanding software evolution, supporting tasks such as code review, change integration, and program comprehension, while also enabling automated adaptation of client code, data-driven refactoring tools, and empirical studies on the impact of refactoring on readability, code smells, and technical debt [22, 38, 43]. However, developers have reported a lack of tool support for integrating refactorings into collaborative workflows, reviewing refactoring-specific edits, and defining new refactoring types [20]. These limitations reduce tool adoption in software maintenance processes.

To address this, the research community has developed a variety of tools that automatically detect refactorings by analyzing version histories [22, 43, 48]. Many of these tools are based on definitions inspired by Fowler's catalog [14]. Among the most prominent are ReExtractor+ [22], RefactoringMiner [48], and RefDiff [43]. ReExtractor+ uses a context-aware statement matching algorithm for Java, RefDiff abstracts language-specific syntax to support JavaScript, C, and Java, and RefactoringMiner applies AST-based algorithms to detect refactoring types in Java, Python, and C++. Despite their value, detecting refactorings remains a non-trivial task [51]. Existing approaches depend on manually crafted rules and complex static analysis techniques, making them difficult to extend or adapt to new refactoring types or programming languages.

The emergence of foundation models offers a promising alternative by enabling code understanding through learned representations and natural language prompts [19, 50, 52]. Fan et al. [13] have highlighted the increasing use of models in computer science, and some open problems in the refactoring area. However, the extent to which these models can reliably detect refactorings, particularly in real-world software, remains underexplored.

This study examines the viability of foundation models—Phi4-14B (Microsoft), Claude 3.5 Sonnet (Anthropic), o4-mini-high (OpenAI), and Gemini 2.5 Pro (Google)—for detecting refactorings using our tool (RefModel). We evaluate these models using a dataset of 858 transformations applied to artificially generated small Java programs, covering widely-used refactoring types. Additionally, we assess their performance on 44 refactorings extracted from four real-world software systems. For comparison, we include the results from three state-of-the-art static-analysis-based tools, namely ReExtractor+, RefactoringMiner, and RefDiff. Our results show that RefModel is competitive with traditional refactoring detection tools. For example, Phi4-14B detects 79.4% of refactorings in small programs and 77.3% in real systems, while Claude 3.5 Sonnet achieves 98.5% and 93.2% on the same tasks, respectively. In both small and real-world settings, Claude 3.5 Sonnet and Gemini 2.5 Pro together correctly identified 100% of the refactorings in the former and 97% in the latter. The models demonstrated consistent performance across varying code sizes. In a cross-language evaluation involving 20 transformations in Python and Golang, Gemini 2.5 Pro detected all refactorings, while Phi4-14B detected 80%. Claude 3.5 Sonnet, o4-mini-high and Gemini 2.5 Pro outperformed the best traditional refactoring detection tools on this dataset.

These findings are encouraging, showing that models not only detect a wide range of refactorings—including those missed by

static-analysis-based tools—but also provide natural language explanations that can improve developer understanding. Unlike traditional tools, RefModel only requires a natural language sentence to define each refactoring type, simplifying extension and evolution.

2 Evaluation: Small Programs

The goal of this evaluation [5] is to assess the effectiveness of foundation models in detecting refactorings from a developer's perspective. We begin with transformations applied to small programs (14–34 LOC) to better isolate and understand the behavior of the models under controlled conditions.

2.1 Research Questions

We address the following research questions (RQs) to achieve the goal of our study:

- RQ_1 To what extent does Phi4-14B detect refactorings?
- RQ_2 To what extent does Claude 3.5 Sonnet detect refactorings?
- RQ_3 How does the performance of foundation models compare to that of traditional refactoring detection tools such as RefactoringMiner?

2.2 Study Design

We conducted our experiments in May 2025, using RefModel with Phi4-14B [10] and Claude 3.5 Sonnet [4], and assessed how varying model sizes influenced detection performance. Phi4-14B was executed locally via the Ollama platform on a MacBook Pro equipped with an M3 processor and 18 GB of RAM. We set the temperature to 0.6 [16], while keeping all other parameters at their default values. Claude 3.5 Sonnet was accessed through its official API (`claude-3-5-sonnet-20241022`) with default settings. All outputs from the models, both in this study and in Section 3, were independently assessed by a minimum of two authors. Disagreements were resolved by involving a third author. We also executed RefactoringMiner 3.0 using its default configuration.

We evaluated ten common refactoring types: Add Parameter, Encapsulate Field, Move Method, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Rename Field, Rename Method, and Rename Class. These refactorings are widely used in practice and frequently cited in the literature [17, 29]. Our evaluation includes both low-level (e.g., Rename Field) and high-level (e.g., Move Method) refactorings.

We evaluate 858 transformations, each representing a single refactoring automatically applied—via scripting—using Eclipse JDT 4.16 to Java programs generated by JDolly [26, 45]. The results produced by the foundation models and RefactoringMiner are compared against this known baseline. Our choice of using Eclipse for applying refactorings is supported by findings from Oliveira et al. [34], who conducted a survey showing that in 4 out of 6 cases, developers preferred the mechanics of Eclipse over those used by traditional detection tools. Our initial dataset consisted of 100 transformations for each refactoring type. We applied automated techniques to filter out incorrect transformations generated by Eclipse that introduced compilation errors or behavioral changes, using both the Java compiler and SafeRefactor [27, 47]. Additionally, previous studies have shown that Eclipse may produce incorrect transformations [33]. For instance, when applying the Pull Up Method or Push Down Method refactorings, the tool occasionally failed to move methods to or from the appropriate superclass or subclass. Such invalid cases were filtered from our dataset using the approach proposed by Oliveira et al. [33].

We use the following prompt to evaluate each transformation applied to the small Java programs. The `definitions` block refers to the refactoring descriptions used in our study, as presented in Table 1. These definitions were inspired by Fowler's catalog [14, 15]. We included more refactoring types than those present in our studies to test the models' robustness and detect potential false positives. The models did not face context window constraints, as the `original` and `refactored` programs were small (14–34 LOC) and could be fully included in the prompt.

> You are an expert coding assistant specialized in software refactoring, with many years of experience analyzing code transformations.
> You will be given two versions of a program:
>
> **Original Version:**
> `original`
>
> **Transformed Version:**
> `refactored`
>
> Your task is to identify which refactoring type(s) have been applied in transforming the original program into the new version. Use only the following list of predefined refactorings:
> `definition`
>
> **Instructions:**
> 1. Begin your response with a bullet-point list of the refactoring type(s) applied.
> 2. Then, briefly justify each identified refactoring with reference to the specific code changes.
> 3. Only include refactorings from the list above.
> 4. Be concise but precise in your explanations.
> Do not generate explanations unrelated to the given transformation.

2.3 Results

Table 2 presents the performance of three refactoring detection approaches, namely Phi4-14B, Claude 3.5 Sonnet, and RefactoringMiner using the metric pass@1 [11]. The pass@1 metric measures whether the first answer produced by a model is correct. The results indicate that both Claude 3.5 Sonnet and RefactoringMiner perform robustly, achieving recall values of 98.5% and 95.8%, respectively. In contrast, Phi4-14B, despite being a significantly smaller model, still achieves a reasonable recall of 79.4%. For precision, Claude 3.5 Sonnet and RefactoringMiner maintain high precision (88.5% and 89.3%, respectively), whereas Phi4-14B lags behind with 56.4%. Certain refactoring types—such as Add Parameter, Encapsulate Field, and Rename Class—are consistently detected by all tools with high accuracy. However, more complex transformations like Move Method and Push Down Method exhibit greater variance across approaches, particularly for Phi4-14B. These results suggest that while Phi4-14B demonstrates potential, especially given its size and efficiency, larger models like Claude 3.5 Sonnet are more reliable across the evaluated transformations. These results suggest that while Phi4-14B can be effective for certain types of refactorings, Claude 3.5 Sonnet and traditional tools still offer more robust performance in scenarios involving class hierarchy and method relocation. The models typically provided clear explanations, referencing relevant code elements and accurately describing the corresponding refactoring.

Table 1: Refactorings and their definitions.

Refactoring	Definition
Add Met. Param.	Introduces a new parameter to an existing method.
Encapsulate Field	Makes a field private and adds getter and setter methods.
Extract Class	Moves a group of related fields and methods from an existing class into a newly created class.
Extract Interface	Creates a new interface from existing method(s) in a class.
Extract Superclass	Creates a new superclass to encapsulate shared attributes and behavior from two or more existing classes.
Inline Class	Merges a class into another when it is too small or redundant.
Inline Method	Replaces a method call with the method's body.
Move Field	Relocates a field to a more appropriate class.
Move Method	Relocates a method to a more appropriate class.
Pull Up Field	Moves a field from a child class to its parent class.
Pull Up Method	Moves a method from a child class to its parent class.
Push Down Field	Moves a field from a parent class to one or more child classes.
Push Down Method	Moves a method from a parent class to one or more child classes.
Remove Method Parameter	Eliminates an existing parameter from a method signature.
Rename Field	Changes the name of a class or instance variable.
Rename Method	Changes the name of a method while preserving its behavior.
Rename Package	Changes the name of a package declaration.
Rename Class	Changes the name of a class without altering its structure.
Rep. Magic Num. with Cons.	Replaces a literal number with a named constant.

Table 2: Performance of each approach on small programs.

Refactoring	Phi4	Claude	RMiner	Total
Add Parameter	97	100	100	100
Encapsulate Field	100	100	100	100
Move Method	43	93	100	100
Pull Up Field	85	100	100	100
Pull Up Method	37	45	45	47
Push Down Field	52	100	100	100
Push Down Method	0	9	3	11
Rename Class	90	99	100	100
Rename Field	65	99	100	100
Rename Method	99	100	74	100
Recall	**79.4%**	**98.5%**	**95.8%**	**100%**
Precision	**56.4%**	**88.5%**	**89.3%**	**100%**

2.4 Discussion

2.4.1 Errors. In some cases, Phi4-14B and Claude 3.5 Sonnet correctly described that a method was pushed down to a subclass. However, both models produced an invalid refactoring name for this operation, referring to it as Pull Down Member. Phi4-14B also failed to distinguish between the more specific Push Down Method and Pull Up Method. Instead, it often generalized the transformation as Move Method. While this is not technically incorrect, we expected the model to recognize and name the more precise hierarchical refactoring involved.

Refactoring mechanics can be customized in various ways, as shown in previous studies [31, 32]. Additionally, our dataset includes several non-trivial transformation scenarios that reflect the complexity and variability encountered in real-world refactoring practices. In one case, a field was pulled from a subclass into its parent class `A`, but `A` already had a field with the same name declared in its parent class. This led Phi4-14B and Claude 3.5 Sonnet to misclassify the transformation as a Push Down Field rather than the correct Pull Up Field. Another challenging example involved pushing down an abstract method: the method was implemented in a subclass, but the abstract declaration in the superclass was not removed. Such cases introduced ambiguity for the models, as they had to infer the intended transformation from partial or inconsistent structural changes.

2.4.2 Prompt. Based on preliminary results, we analyzed initial errors and refined the prompt. For example, we found that some of the definitions presented in Table 1 were imprecise or overly similar. In particular, the definitions for Move Method, Pull Up Method, and Push Down Method lacked clarity regarding class hierarchies. We revised them to explicitly describe the direction of transformation in the inheritance structure. Similarly, we distinguished Extract Class from Extract Superclass by emphasizing the number of classes involved and the structural context of the transformation. To further improve prompt quality, we applied a technique known as *metaprompting* [40], in which a foundation model is used to enhance the design of the original prompt. Specifically, we used o4-mini-high to revise and optimize the instructions.

2.4.3 Gemini 2.5 Pro. We investigated whether Gemini 2.5 Pro could address the cases missed by Claude 3.5 Sonnet by re-running it on those transformations. Remarkably, Gemini 2.5 Pro correctly identified all of them.

2.5 Threats to Validity

Some validity must be considered in studies involving foundation models for software engineering tasks [42]. One potential threat to internal validity concerns the accuracy and consistency of our experimental setup, particularly the design of prompts used to query the foundation models. Although we made efforts to craft prompts that were neutral and uniform across evaluations, slight variations in wording may still influence model outputs and affect comparability. Another threat lies in our use of artificially generated small programs, created using JDolly, to apply individual refactoring transformations. This strategy allows for controlled experimentation and precise isolation of each refactoring type. However, such synthetic code does not fully capture the complexity, coding conventions, or contextual dependencies present in real-world software. Consequently, the behavior of foundation models on these examples may not generalize to more realistic development environments. To mitigate this threat, we complemented our evaluation with a dataset of refactorings applied to real-world systems, as follows.

3 Evaluation: Real Programs

The goal of this evaluation is to assess the effectiveness of foundation models in detecting refactorings from a developer's perspective, focusing on transformations applied to real programs.

Table 3: Java projects used for refactoring evaluation.

Project	Domain	KLOC	Stars	Contrib.	Transf.
Lettuce	A scalable thread-safe Redis client for synchronous, asynchronous and reactive usage.	234K	5.6	135	30
Apache Gobblin	A distributed data integration framework.	454K	2.6	115	1
Google Maps Services	A Java client for Google Maps Services.	38K	1.7	96	3
Spring Boot	A framework to create Spring-based applications.	674K	77.1	1,156	4
RefMiner	A refactoring detection tool.	127K	0.4	18	6

3.1 Research Questions

We address the following RQs to achieve the goal:

- RQ_1 To what extent does Phi4-14B detect refactorings?
- RQ_2 To what extent do Claude 3.5 Sonnet, Gemini 2.5 Pro, o4-mini-high detect refactorings?
- RQ_3 How does the performance of foundation models compare to that of traditional refactoring detection tools such as ReExtractor+, RefactoringMiner, and RefDiff?

3.2 Study Design

We conducted our experiments in May 2025, using RefModel with Phi4-14B, Claude 3.5 Sonnet, Gemini 2.5 Pro [18], and o4-mini-high [37]. For Phi4-14B, Claude 3.5 Sonnet, and RefactoringMiner, we adopted the same setup described in Section 2.2. Additionally, Gemini 2.5 Pro and o4-mini-high were manually executed via their official web interfaces using default parameters. These models are highly ranked on the Chatbot Arena leaderboard [1]. We also included ReExtractor+ (version 2.4.4) and RefDiff (version 2.0) in our evaluation, both executed with their default configurations.

We evaluated 44 transformations applied to real-world open-source Java projects (see Table 3) using IntelliJ (version 2024.1.4), covering 12 widely used refactoring types: Extract Class, Extract Interface, Extract Superclass, Inline Method, Move Method, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Rename Field, Rename Method, and Rename Class. These transformations were selected based on their prevalence in real-world projects [17, 29]. Each transformation consists of a single refactoring instance manually applied using IntelliJ, allowing for a more precise and controlled analysis of each approach's performance. The results produced by the foundation models and static-analysis-based tools are compared against this baseline. We use the following prompt when evaluating transformations applied to real programs:

> You are an expert coding assistant specialized in software refactoring, with many years of experience analyzing code transformations.
> You will be given the diffs of a commit:
>
> **Diffs:**
> `diff`
>
> Your task is to identify which refactoring type(s) have been applied in transforming the original program into the new version. Use only the following list of predefined refactorings:
> `definition`
>
> **Instructions:**
> 1. Begin your response with a bullet-point list of the refactoring type(s) applied.
> 2. Then, briefly justify each identified refactoring with reference to the specific code changes.
> 3. Only include refactorings from the list above.
> 4. Be concise but precise in your explanations.
> Do not generate explanations unrelated to the given transformation.

Table 4: Performance on real-world programs.

Refactoring	Phi4	Claude	Gemini		RMiner	RefDiff	ReExt.	Total
Extract Class	100%	0%	0%	0%	100%	100%	100%	1
Extract Interface	100%	100%	75%	100%	100%	100%	75%	8
Extract Superclass	100%	100%	100%	100%	100%	100%	100%	2
Inline Method	80%	100%	100%	100%	60%	40%	60%	5
Move Method	38%	100%	100%	87.5%	75%	100%	87.5%	8
Pull Up Field	100%	100%	100%	100%	100%	0%	100%	2
Pull Up Method	80%	80%	80%	100%	100%	100%	100%	5
Push Down Field	0%	100%	100%	100%	100%	100%	100%	1
Push Down Meth.	50%	75%	100%	100%	75%	50%	75%	4
Rename Class	100%	100%	100%	100%	100%	100%	100%	4
Rename Field	100%	100%	100%	66.6%	100%	100%	100%	1
Rename Method	100%	100%	100%	100%	100%	100%	100%	3
Recall	**79.7%**	**92.2%**	**93.8%**	**93.8%**	**88.6%**	**77.3%**	**84.1%**	**100%**
Precision	**61.4%**	**77.6%**	**82.2%**	**81.1%**	**41.9%**	**70.8%**	**40.2%**	**100%**

To deal with context-window [12] limitations, we analyze the GitHub `diff`, which includes the added, removed and some unchanged lines of code, in the prompt. `definitions` indicates the refactoring definitions used in our work (Table 1).

3.3 Results

Table 4 presents the detection accuracy of foundation models (Phi4-14B, Claude 3.5 Sonnet, Gemini 2.5 Pro, o4-mini-high) and static-analysis-based refactoring detection tools (ReExtractor+, RefactoringMiner, RefDiff) on a dataset of 44 using the metric pass@1 [11]. Each cell shows the percentage of refactorings correctly identified. The last column indicates the number of evaluated instances. Overall, Gemini 2.5 Pro and o4-mini-high reached the highest recall scores, both with 93.8%, followed closely by Claude 3.5 Sonnet (92.2%) and RefactoringMiner (88.6%). Phi4-14B obtained a recall of 79.7%, while RefDiff and ReExtractor+ reached 77.3% and 84.1%, respectively. For precision, Gemini 2.5 Pro led with 82.2%, followed closely by o4-mini-high (81.1%) and Claude 3.5 Sonnet (77.6%). Phi4-14B reached 61.4%, while traditional tools RefactoringMiner and ReExtractor+ had lower precision values of 41.9% and 40.2%, respectively. RefDiff stood at 70.8%. We also ran Claude 4 Sonnet, and it achieved the same results as Claude 3.5 Sonnet. Notably, Claude 3.5 Sonnet, Gemini 2.5 Pro, and o4-mini-high consistently reach perfect or near-perfect scores across most refactorings, while Phi4-14B showed strong results on simpler transformations (e.g., Rename Method) but struggled with structural ones such as Move Method and Push Down Field.

3.4 Discussion

3.4.1 Errors. The types of errors observed were similar to those discussed in Section 2.4.1. Claude 3.5 Sonnet showed recurring issues related to confusion between Pull Up and Push Down refactorings. Although it often identified the source and destination of the change correctly, it mislabeled the direction when there was no clear class hierarchy. Furthermore, Claude 3.5 Sonnet occasionally confused Move and Rename refactorings, especially when a method or field was renamed to an identifier that already existed in another class, leading it to infer relocation instead of renaming.

Gemini 2.5 Pro presented confusion between structurally similar refactorings. In one case, it misclassified an Extract Class as an Extract Superclass, failing to distinguish between horizontal and hierarchical decomposition. In another instance, it incorrectly described the creation of a new super-interface. Additionally, Gemini

Table 5: Recall (%) by diff size range (LOC) across approaches.

Diff	Phi4	Claude	Gemini		RMiner	RefDiff	ReExt.	Total
0–39	80%	100%	100%	100%	90%	30%	70%	10
40–79	81%	86%	81%	86%	86%	76%	86%	21
80–119	83%	100%	100%	100%	100%	67%	100%	6
120–159	25%	100%	100%	100%	75%	25%	75%	4
160–359	100%	100%	100%	100%	100%	0%	100%	3

2.5 Pro struggled to identify Pull Up Method refactorings when the class hierarchy was not visible in the GitHub diff. Without explicit inheritance information, the model could not infer the correct direction of the transformation.

Phi4-14B exhibited similar difficulties. One frequent error was confusing the direction of refactorings, such as using incorrect labels like "push up" or "pull down" when referring to Pull Up or Push Down Method refactorings. In transformations with multiple classes with similarly named methods, Phi4-14B often failed to correctly identify which class the method was moved from or to. Like Claude 3.5 Sonnet, it also misclassified Rename operations as Move refactorings when the new name matched an existing identifier in another class. In cases where class hierarchy information was absent, Phi4-14B similarly struggled to detect Pull Up refactorings.

o4-mini-high was able to detect class hierarchies and correctly identify Push Down and Pull Up refactorings that were missed by Claude 3.5 Sonnet and Gemini 2.5 Pro, demonstrating its ability to infer structural relationships in the code. These findings underscore the importance of structural context in refactoring detection. Foundation models remain sensitive to missing or incomplete information—particularly in transformations involving class hierarchies (e.g., Pull Up and Push Down) or subtle naming conflicts (e.g., Rename vs. Move). In such cases, the absence of explicit inheritance or semantic overlap can easily lead to misclassification.

3.4.2 Granularity. Table 5 presents the detection recall of various foundation models and traditional tools across different diff size (granularity) ranges, measured in LOC. Each row corresponds to a range of diff sizes, and each cell reports the percentage of correctly detected refactorings for that range. The last column indicates the total number of evaluated transformations. The results show that most approaches, particularly Claude 3.5 Sonnet, Gemini 2.5 Pro, and RefactoringMiner, maintain high accuracy regardless of diff size, with perfect or near-perfect scores across all ranges. In contrast, Phi4-14B exhibits reduced performance for medium-sized diffs (120–159 LOC), while RefDiff struggles with both small and large diffs, reaching as low as 0% in the 160–359 LOC range.

3.4.3 Evolution. Evolving our tool to refine existing refactoring definitions or to support new refactoring types requires only updating or adding natural language definitions—such as those shown in Table 1. This process is significantly simpler and more flexible than implementing the complex rules and static analysis algorithms required by traditional tools [9, 30, 39, 43, 48]. Furthermore, it enables interactive exploration of refactorings, allowing developers to query, verify, or refine the detected changes.

3.4.4 Larger Programs. Foundation models still face limitations related to context window length [12]. To address this, we designed a prompt that focuses specifically on the code transformation—rather than the entire program—enabling the models to operate effectively within the available context. Although the largest `diff` in our dataset contained 359 lines, it remained within the context window limits of the evaluated models. Our results show that foundation models can detect most refactorings and perform on par with, or slightly better than, the best static-analysis-based tools. Notably, foundation models are evolving rapidly, with newer versions offering substantially larger context windows, further expanding their applicability to software engineering tasks. As future work, we plan to investigate the use of retrieval-augmented generation (RAG) to further enhance performance in large codebases, following recent work by Batole et al. [6] that applied this strategy to automate the Move Method refactoring.

3.4.5 Other Languages. As a feasibility study, we extended our evaluation to include programming languages beyond Java. Using the same experimental setup, we applied 20 refactorings to three open-source projects, namely `zap`, `examples`, and `full-stack-fastapi-template`, covering 10 transformations in Python and 10 in Golang. The selected transformations produced git diffs of up to 149 lines of code. Refactorings were applied using the PyCharm 2025.1.1.1 and IntelliJ 2024.1.4 IDEs, depending on the language and project. Phi4-14B detected 80% of the transformations, while Claude 3.5 Sonnet, o4-mini-high, and Gemini 2.5 Pro achieved 90%, 95%, and 100%, respectively, highlighting their promising potential for multi-language refactoring detection.

3.5 Threats to Validity

We acknowledge similar threats to validity as those discussed in Section 2.5. A potential threat to internal validity concerns the correctness of the refactorings applied in our dataset. Although we used IntelliJ to automate the transformations, some refactorings resulted in compilation errors, behavioral changes, or were incorrectly applied [33]. To ensure the reliability of our ground truth, we manually validated and excluded all such faulty cases. To mitigate threats to external validity, we included 64 transformations from four real-world systems. However, the generalizability of our findings to larger and more diverse codebases remains a limitation. Additionally, we focused our analysis on smaller, isolated refactorings. Construct validity may also be affected by ambiguities in prompt phrasing or model interpretation. For instance, despite adopting a clear and consistent taxonomy (Table 1), some misclassifications occurred due to non-standard terminology or overly generic responses from the models. Lastly, although our results show that foundation models can effectively detect refactorings, particularly in real-world scenarios, the size of our dataset may limit broader statistical conclusions. We did not conduct formal hypothesis testing.

4 Related Work

Dig et al. [9] proposed an algorithm for detecting refactorings applied during component evolution. Their approach was implemented as an Eclipse plugin called RefactoringCrawler. The tool was evaluated on three software components, ranging in size from 17 KLOC to 352 KLOC, and reached over 85% accuracy across seven types of refactorings. Prete et al. [39] developed Ref-Finder, a tool that detects refactorings using a template-based reconstruction approach. Ref-Finder is capable of identifying 63 out of the 72

refactoring types described in Fowler's catalog [14]. The evaluation reported a precision of 79% and a recall of 95%. Soares et al. [46] conducted a comparative study of three refactoring detection approaches: (i) manual inspection [29], (ii) commit message analysis [41], and (iii) dynamic analysis using SafeRefactor [27, 28, 47]. Their analysis considered behavioral preservation as a criterion and revealed that Ref-Finder exhibited low precision and recall.

Silva et al. [43] introduced RefDiff 2.0, a language-agnostic refactoring detection tool. It incorporates a novel algorithm based on Code Structure Trees, which abstracts away language-specific syntax, enabling support for multiple programming languages such as Java, C, and JavaScript. Tsantalis et al. [48] developed RefactoringMiner 2.0, an AST-based tool that detects over 100 refactoring types in Java without requiring manually defined thresholds. Later, Alikhanifard and Tsantalis [2] improved the tool's statement mapping capabilities, releasing RefactoringMiner 3.0 with enhanced accuracy. To evaluate these tools, the authors executed RefactoringMiner 2.0, GumTreeDiff, and two versions of RefDiff on hundreds of commits from open-source GitHub projects, monitored over a two-month period using an existing dataset [49]. The union of all true positives detected by at least one tool was used as ground truth. Manual validation was conducted by two authors, resulting in 7,226 confirmed refactoring instances across 40 refactoring types. RefDiff also leveraged this dataset [49] to evaluate its precision and recall, supplementing it with additional manually curated instances. We plan to expand our evaluation to incorporate this dataset, enabling a broader and more comprehensive analysis of model performance and detection accuracy.

Liu et al. [22, 23] proposed ReExtractor+, a refactoring detection technique designed to identify both high-level and low-level refactorings. It leverages a reference-based entity matching algorithm that uses qualified names, implementation details, and reference information to match coarse-grained code entities across consecutive program versions. It incorporates a context-aware statement matching algorithm. An evaluation on real-world systems shows that ReExtractor+ reduces false positives by 59.6% and improves recall by 19.2% over existing tools.

Leandro et al. [21] proposed a technique for systematically testing refactoring detection tools. To evaluate their approach, they automatically applied 9,885 transformations across four real-world open-source projects using 8 refactorings supported by the Eclipse IDE. The authors reported 34 issues to the developers of RefactoringMiner and RefDiff. Oliveira et al. [34] conducted a survey with 53 developers from popular Java projects on GitHub to investigate whether the mechanics used by refactoring detection tools—such as RefactoringMiner [48] and RefDiff [43]—align with developer expectations in practice. The results revealed that these tools often fail to detect many refactorings that developers consider relevant. In four out of six scenarios presented, the majority of participants expressed a preference for the refactoring behavior implemented by the Eclipse IDE over that of the refactoring detection tools.

In this work, we introduce RefModel, which uses foundation models to detect code refactorings, unlike previous approaches based on static analysis. We evaluate it on two novel datasets—one synthetic and one real-world—not used in prior work. In both datasets, RefModel performs on par with, or slightly better than, the state-of-the-art tools such as ReExtractor+, RefactoringMiner, and RefDiff. Foundation models like Claude 3.5 Sonnet and Gemini 2.5 Pro demonstrate strong potential as a flexible, language-independent alternative for supporting refactoring detection in modern software development. Our approach requires only a single natural language sentence to define each refactoring type, greatly simplifying its extension and evolution.

5 Conclusion

In this paper, we evaluate the ability of foundation models to detect code refactorings using our tool RefModel, and compare them against traditional static analysis tools. Using a dataset of transformations applied to both synthetic and real-world Java programs, we assessed the capabilities of Phi4-14B, Claude 3.5 Sonnet, o4-mini-high and Gemini 2.5 Pro, benchmarking them against RefactoringMiner and RefDiff. Our results show that foundation models—particularly Claude 3.5 Sonnet, o4-mini-high, and Gemini 2.5 Pro—reach high recall and precision, in some cases outperforming the state-of-the-art static tools. Across the 64 refactorings applied to real-world systems, Claude 3.5 Sonnet and Gemini 2.5 Pro jointly reached a detection recall of 97%. Despite being a smaller model, Phi4-14B also performed competitively across several refactoring types. Some misclassifications were observed, often due to limitations in the available context (e.g., missing class hierarchies) or ambiguities between similar refactoring types.

Foundation models are a viable and flexible alternative to traditional refactoring detection tools, particularly in scenarios where static analysis is difficult to apply or extend. Unlike rule-based tools, our approach requires only a single natural language definition of each refactoring type, significantly reducing the effort needed to support new transformations. As foundation models continue to evolve—with larger context windows, improved precision, and advancements such as retrieval-augmented generation—they are likely to become even more effective, interpretable, and adaptable across programming languages and development environments. While promising, foundation models still involve trade-offs related to runtime and cost. We ran Phi4-14B locally on modest hardware, with each analysis completing in just a few seconds. In contrast, using LLMs incurs API-related expenses and requires sending code to the cloud, potentially raising concerns around latency and privacy.

As future work, we plan to expand our evaluation by including non-refactoring transformations, a broader set of refactoring types, and support for more programming languages. We also aim to investigate refactorings interleaved with other code changes [29], as well as alternative models, agentic approaches [3, 24], and diverse prompting strategies. In addition to applying refactorings using other IDEs, we will explore the detection of composite refactorings—those involving multiple operations [7, 8]—using retrieval-augmented generation techniques [6].

ARTIFACT AVAILABILITY

All study artifacts are available online [44].

ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work was partially supported by CNPq (403719/2024-0, 310313/2022-8, 404825/2023-0, 443393/2023-0, 312195/2021-4), FAPESQ-PB (268/2025).

REFERENCES

[1] 2025. Chatbot Arena LLM Leaderboard. https://lmarena.ai.

[2] Pouria Alikhanifard and Nikolaos Tsantalis. 2025. A Novel Refactoring and Semantic Aware Abstract Syntax Tree Differencing Tool and a Benchmark for Evaluating the Accuracy of Diff Tools. *Transactions on Software Engineering and Methodology* 34, 2 (2025).

[3] Anthropic. 2024. Building effective agents. https://www.anthropic.com/research/building-effective-agents

[4] Anthropic. 2024. Claude 3.5. https://www.anthropic.com/news/claude-3-5-sonnet/.

[5] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. *The Goal Question Metric Approach*. 528–532 pages.

[6] Fraol Batole, Abhiram Bellur, Malinda Dilhara, Mohammed Raihan Ullah, Yaroslav Zharov, Timofey Bryksin, Kai Ishikawa, Haifeng Chen, Masaharu Morimoto, Shota Motoura, Takeo Hosomi, Tien N. Nguyen, Hridesh Rajan, Nikolaos Tsantalis, and Danny Dig. 2025. Leveraging LLMs, IDEs, and Semantic Embeddings for Automated Move Method Refactoring. arXiv:2503.20934 [cs.SE] https://arxiv.org/abs/2503.20934

[7] Ana Bibiano, Wesley Assunção, Daniel Coutinho, Kleber Santos, Vinícius Soares, Rohit Gheyi, Alessandro Garcia, Baldoino Fonseca, Márcio Ribeiro, Daniel Oliveira, Caio Barbosa, João Marques, and Anderson Oliveira. 2021. Look Ahead! Revealing Complete Composite Refactorings and their Smelliness Effects. In *International Conference on Software Maintenance and Evolution*. 298–308.

[8] Ana Bibiano, Vinicius Soares, Daniel Coutinho, Eduardo Fernandes, João Lucas Correia, Kleber Santos, Anderson Oliveira, Alessandro Garcia, Rohit Gheyi, Baldoino Fonseca, Márcio Ribeiro, Caio Barbosa, and Daniel Oliveira. 2020. How Does Incomplete Composite Refactoring Affect Internal Quality Attributes?. In *International Conference on Program Comprehension*. 149–159.

[9] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *European Conference on Object-Oriented Programming*. 404–428.

[10] Marah Abdin et al. 2024. Phi-4 Technical Report. https://arxiv.org/abs/2412.08905

[11] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374

[12] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*.

[13] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *International Conference on Software Engineering: Future of Software Engineering*. IEEE, 31–53.

[14] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley.

[15] Martin Fowler. 2025. Catalog of Refactorings. https://refactoring.com/catalog/.

[16] Rohit Gheyi, Marcio Ribeiro, and Jonhnanthan Oliveira. 2025. Evaluating the Effectiveness of Small Language Models in Detecting Refactoring Bugs. arXiv:2502.18454 [cs.SE] https://arxiv.org/abs/2502.18454

[17] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: a large-scale survey of software refactoring. In *Foundations of Software Engineering*. 1303–1313.

[18] Google. 2025. Gemini 2.5 Pro. https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/.

[19] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024).

[20] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Foundations of Software Engineering*. 50:1–50:11.

[21] Osmar Leandro, Rohit Gheyi, Leopoldo Teixeira, Márcio Ribeiro, and Alessandro F. Garcia. 2022. A Technique to Test Refactoring Detection Tools. In *Brazilian Symposium on Software Engineering*. 188–197.

[22] Bo Liu, Hui Liu, Nan Niu, Yuxia Zhang, Guangjie Li, He Jiang, and Yanjie Jiang. 2025. An Automated Approach to Discovering Software Refactorings by Comparing Successive Versions. *IEEE Transactions on Software Engineering* 51, 5 (2025), 1358–1380.

[23] Bo Liu, Hui Liu, Nan Niu, Yuxia Zhang, Guangjie Li, and Yanjie Jiang. 2023. Automated Software Entity Matching Between Successive Versions. In *Automated Software Engineering*. 1615–1627.

[24] Rian Melo, Pedro Simões, Rohit Gheyi, Marcelo d'Amorim, Márcio Ribeiro, Gustavo Soares, Eduardo Almeida, and Elvys Soares. 2025. Agentic SLMs: Hunting Down Test Smells. arXiv:2504.07277 [cs.SE] https://arxiv.org/abs/2504.07277

[25] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.

[26] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. 2018. Detecting Overly Strong Preconditions in Refactoring Engines. *IEEE Transactions on Software Engineering* 44, 5 (2018), 429–452.

[27] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. 2014. Making refactoring safer through impact analysis. *Science of Computer Programming* 93 (2014), 39–64.

[28] Melina Mongiovi, Gustavo Mendes, Rohit Gheyi, Gustavo Soares, and Márcio Ribeiro. 2014. Scaling Testing of Refactoring Engines. In *International Conference on Software Maintenance and Evolution*. 371–380.

[29] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How we refactor, and how we know it. In *International Conference on Software Engineering*. IEEE, 287–297.

[30] Stas Negara, Nicholas Chen, M. Vakilian, Ralph Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *European Conference on Object-Oriented Programming*. 552–576.

[31] Daniel Oliveira, Wesley K. G. Assunção, Alessandro F. Garcia, Ana Carla Bibiano, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. 2023. The untold story of code refactoring customizations in practice. In *International Conference on Software Engineering*. 108–120.

[32] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. 2019. Revisiting the Refactoring Mechanics. *Information and Software Technology* 110 (2019), 136–138.

[33] Jonhnanthan Oliveira, Rohit Gheyi, Felipe Pontes, Melina Mongiovi, Márcio Ribeiro, and Alessandro Garcia. 2020. Revisiting Refactoring Mechanics from Tool Developers' Perspective. In *Brazilian Symposium on Formal Methods*. 25–42.

[34] Jonhnanthan Oliveira, Rohit Gheyi, Leopoldo Teixeira, Márcio Ribeiro, Osmar Leandro, and Baldoino Fonseca. 2023. Towards a better understanding of the mechanics of refactoring detection tools. *Information and Software Technology* 162 (2023), 107273.

[35] William Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[36] William Opdyke and Ralph Johnson. 1990. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Symposium Object-Oriented Programming Emphasizing Practical Applications*. 274–282.

[37] OpenAI. 2025. OpenAI o3-mini. https://openai.com/index/openai-o3-mini/.

[38] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An Exploratory Study on the Relationship between Changes and Refactoring. In *International Conference on Program Comprehension*. 176–185.

[39] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based Reconstruction of Complex Refactorings. In *International Conference on Software Maintenance*. 1–10.

[40] PromptHub. 2025. A Complete Guide to Meta Prompting.

[41] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. 2008. On the relation of refactorings and software defect prediction. In *Mining Software Repositories*. 35–38.

[42] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the silence: the threats of using llms in software engineering. In *International Conference on Software Engineering: New Ideas and Emerging Results*. 102–106.

[43] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802.

[44] P. Simões, R. Gheyi, R. Melo, J. Oliveira, M. Ribeiro, and W. Assunção. 2025. RefModel: Detecting Refactorings using Foundation Models (artifacts). https://github.com/brain-ufcg/RefModel/.

[45] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2013. Automated Behavioral Testing of Refactoring Engines. *IEEE Transactions on Software Engineering* 39, 2 (2013), 147–162.

[46] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. 2013. Comparing Approaches to Analyze Refactoring Activity on Software Repositories. *Journal of Systems and Software* 86, 4 (2013), 1006–1022.

[47] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making Program Refactoring Safer. *IEEE Software* 27, 4 (2010), 52–57.

[48] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930 – 950.

[49] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *International Conference on Software Engineering (ICSE)*. 483–494.

[50] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50 (2024), 911–936.

[51] Peter Weissgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *Automated Software Engineering*. 231–240.

[52] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *International Symposium on Machine Programming*. 1–10.