Automated Software Engineering (2025) 32:26
https://doi.org/10.1007/s10515-025-00500-0

Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study

Bo Liu[1] · Yanjie Jiang[1,2] · Yuxia Zhang[1] · Nan Niu[3] · Guangjie Li[4] · Hui Liu[1]

Received: 19 November 2024 / Accepted: 12 February 2025

Abstract

Software refactoring is an essential activity for improving the readability, maintainability, and reusability of software projects. To this end, a large number of automated or semi-automated approaches/tools have been proposed to locate poorly designed code, recommend refactoring solutions, and conduct specified refactorings. However, even equipped with such tools, it remains challenging for developers to decide where and what kind of refactorings should be applied. Recent advances in deep learning techniques, especially in large language models (LLMs), make it potentially feasible to automatically refactor source code with LLMs. However, it remains unclear how well LLMs perform compared to human experts in conducting refactorings automatically and accurately. To fill this gap, in this paper, we conduct an empirical study to investigate the potential of LLMs in automated software refactoring, focusing on the identification of refactoring opportunities and the recommendation of refactoring solutions. We first construct a high-quality refactoring dataset comprising 180 real-world refactorings from 20 projects, and conduct the empirical study on the dataset. With the to-be-refactored Java documents as input, ChatGPT and Gemini identified only 28 and 7 respectively out of the 180 refactoring opportunities. The evaluation results suggested that the performance of LLMs in identifying refactoring opportunities is generally low and remains an open problem. However, explaining the expected refactoring subcategories and narrowing the search space in the prompts substantially increased the success rate of ChatGPT from 15.6 to 86.7%. Concerning the recommendation of refactoring solutions, ChatGPT recommended 176 refactoring solutions for the 180 refactorings, and 63.6% of the recommended solutions were comparable to (even better than) those constructed by human experts. However, 13 out of the 176 solutions suggested by ChatGPT and 9 out of the 137 solutions suggested by Gemini were unsafe in that they either changed the functionality of the source code or introduced syntax errors, which indicate the risk of LLM-based refactoring.

Extended author information available on the last page of the article

Published online: 01 March 2025

Keywords Software refactoring · Large language model · Empirical study · Software quality

1 Introduction

Software refactoring is widely employed to improve software quality, especially its readability, maintainability, and reusability (Fowler 1999; Mens and Tourwé 2004; Baqais and Alshayeb 2020). To facilitate software refactoring, a large number of approaches/tools have been proposed to identify refactoring opportunities (Tourwé and Mens 2003; Tsantalis and Chatzigeorgiou 2009, 2011), to recommend refactoring solutions (Mkaouer et al. 2017; Alizadeh and Kessentini 2018; Alizadeh et al. 2020), and to automatically conduct specified refactorings (Ge et al. 2012; Foster et al. 2012; Alizadeh et al. 2019). Refactoring engines like IntelliJ IDEA (JetBrains 2024) and JDeodorant (Fokaefs et al. 2007; Tsantalis et al. 2008) have been successfully adopted to automate the execution of refactorings. However, before the refactorings could be executed automatically, developers should explicitly specify which part of the source code should be refactored, what kind of refactorings should be applied, and the detailed parameters of the refactorings (e.g., a new method name for *rename method* refactoring). Even with the support of the state-of-the-art refactoring approaches/tools, it remains challenging and time-consuming for developers to make such decisions (Feitelson et al. 2022; Peruma et al. 2022), which in turn prevents software refactoring from reaching its maximal potential.

Large language models (LLMs), like ChatGPT (OpenAI 2024a) and Gemini (Team et al. 2023), have the ability to learn complex and massive knowledge (Liu et al. 2023). Consequently, LLMs emerge as a potential solution that could significantly advance automated software refactoring. It is reported that LLMs have demonstrated promising results in various software engineering tasks, like code generation (Mu et al. 2023), fault location (Wu et al. 2023), and program repair (Xia and Zhang 2023). LLMs' impressive capability in understanding and generating natural languages and source code makes it potentially feasible to automatically refactor source code with LLMs. However, it remains unclear how well LLMs perform in automated software refactoring, whether the refactorings suggested/conducted by LLMs are of high quality, and whether refactorings conducted by LLMs are reliable.

To fill this gap, in this paper, we conduct a comprehensive empirical study to obtain a profound insight into the challenges and opportunities of LLM-based refactoring as well as to understand the extent to which LLMs can automate software refactoring to alleviate the burden on developers. We select two representative LLMs, i.e., ChatGPT and Gemini, for the study, and answer the following two research questions:

- RQ1: How well do LLMs (ChatGPT and Gemini) work in the identification of refactoring opportunities?
- RQ2: How well do ChatGPT and Gemini work in the recommendation of refactoring solutions?

To answer the above questions, we first construct a high-quality refactoring dataset comprising 180 real-world refactorings from 20 projects. After that, we request LLMs to refactor the entire Java documents that contain the discovered refactoring opportunities. We conduct a quantitative analysis of LLMs's effectiveness in identifying refactoring opportunities. We also request three human experts to manually and independently assess the quality of the refactoring solutions suggested by LLMs. Our evaluation results suggest that LLMs have the potential for automated software refactoring. However, their performance varies significantly among different types of refactorings, and thus we refine their performance by a taxonomy of refactorings. We also noticed that LLM-based refactoring could be risky. 22 out of the 313 solutions suggested by ChatGPT and Gemini were unsafe in that they either changed the functionality of the source code or introduced syntax errors.

In this paper, we make the following contributions:

- **A comprehensive empirical study** on evaluating LLMs' potential in automated software refactoring. It reveals the strengths and weaknesses of LLMs in software refactoring, refined by a refactoring taxonomy.
- **A new high-quality refactoring dataset** validated with multiple tools and refactoring experts, which could serve as a valuable resource for future research in this area.

The rest of the paper is structured as follows. Section 2 details the study design. Section 3 evaluates the capability of LLMs in identifying refactoring opportunities. Section 4 evaluates the capability of LLMs in recommending refactoring solutions. Section 5 delves into the threats to validity, limitations, and implications. Section 6 provides an overview of related work. Finally, Sect. 7 concludes the paper and suggests future directions.

2 Study design

2.1 Research questions

The empirical study is designed to evaluate the capabilities of large language models (LLMs) in automated software refactoring. Specifically, the study investigates the following research questions:

RQ1. How well do LLMs work in the automated identification of refactoring opportunities?

RQ2. How well do LLMs work in the automated recommendation of refactoring solutions?

In this study, we employ GPT-4 (version: gpt-4-1106-preview) (Achiam et al. 2023; OpenAI 2024b) (called GPT for short in the rest of this paper) and Gemini-1.0 Pro (Google 2024) (called Gemini for short) as the evaluated large language models because they represent the state of the art in this area (Minna et al. 2024; DePalma et al. 2024; Pomian et al. 2024a, b) and refactoring. RQ1 concerns the capability of LLMs in refactoring opportunity identification. RQ1 can be further divided into four sub-research questions:

RQ1-1. How well do LLMs work in identifying refactoring opportunities without specifying concrete refactoring types?

RQ1-2. How well do LLMs work in identifying refactoring opportunities when refactoring types are explicitly specified?

RQ1-3. How does the size of the to-be-refactored source code influence the performance of LLMs in identifying refactoring opportunities?

RQ1-4. In which cases do LLMs perform well or poorly in identifying refactoring opportunities? And how can prompt strategies be improved to identify more refactoring opportunities?

By comparing RQ1-1 and RQ1-2, we can not only reveal the potential of LLMs, but also reveal to what extent explicitly specifying refactoring types can improve the performance of LLM-based refactorings. Notably, when the to-be-refactored source code is longer and more complex, it could be more challenging for LLMs to pick up the expected refactoring opportunities. To validate this hypothesis, we investigate RQ1-3 to reveal the correlation between the performance of LLM-based refactoring opportunity identification and the size of the to-be-refactored source code. By answering RQ1-4, we can gain a deeper understanding of the strengths and weaknesses of LLMs in identifying refactoring opportunities and whether the proposed prompt strategies can facilitate LLMs to identify more refactoring opportunities.

RQ2 concerns the capability of LLMs to suggest refactoring solutions. It can be further divided into two sub-research questions:

RQ2-1. How do the refactoring solutions suggested by LLMs compare to those conducted by human experts?

RQ2-2. Are the suggested refactoring solutions safe? If not, how often are the solutions unsafe, and what is the problem with the suggested solutions?

By answering RQ2-1, we may reveal the potential of LLMs in recommending refactoring solutions. Notably, as generic large language models do not validate the equivalence of the input (original source code) and output (source code after

refactoring), it is likely that the output could be semantically nonequivalent to the input, which results in changes in software system's external behaviors. However, according to the definition of software refactoring (Fowler 1999), software refactoring should never change external behaviors. In this case, the suggested "refactorings" are unsafe. Answering RQ2-2 would reveal how often the refactoring solutions suggested by LLMs are unsafe and why they are unsafe.

2.2 Dataset

To conduct our study, we constructed a new high-quality refactoring dataset, called *ref-Dataset*, due to the following reason: The existing datasets might have been utilized to train the selected LLMs. The new dataset was constructed to better evaluate the generalization capabilities of LLMs. To this end, we reused the 20 popular Java projects chosen by Grund et al. (2021). These projects contained rich evolution histories from the year 2000 to the present. They were from different domains (including code analysis, search engine, unit testing, etc.), which might help reduce the potential bias in the evaluation. Finally, they are publicly available, which facilitates the reproduction of our evaluation. However, project `junit4` did not have any code commits after April 2023, which indicates that all their data (including refactoring activities) may have been used as training data for GPT. Consequently, it was not used in our study. Notably, project `lucene-solr` has been split into two separate projects `lucene` and `solr`, and thus we used the two sub-projects in our study.

From the selected projects, we discovered refactorings conducted on such projects. It is worth noting that manually identifying refactoring operations performed by developers from the commit history is a tedious, time-consuming, and error-prone task. To address this challenge, numerous refactoring detection tools (Dig et al. 2006; Xing and Stroulia 2008; Prete et al. 2010; Kim et al. 2010; Silva and Valente 2017; Silva et al. 2020; Tsantalis et al. 2018, 2022; Liu et al. 2023) have been proposed to automatically discover refactorings. We applied the state-of-the-art refactoring detection tools (i.e., `ReExtractor` (Liu et al. 2023; Liu 2024a) and `RefactoringMiner` (Tsantalis et al. 2018, 2022)) independently to mine refactorings starting from the latest commit of each project. These two tools were selected because (1) they represented the state of the art in refactoring detection; and (2) they supported the detection of both high-level refactorings (e.g., *rename method*) and low-level refactorings (e.g., *extract variable*) at commit level. Notably, we mined refactorings from all commits submitted after April 2023 because GPT's training data extends up to this date (OpenAI 2024b), while Gemini's training data extends to February 2023 (Google 2024). Considering the capability of the employed refactoring detection tools and the popularity of refactorings (Murphy-Hill et al. 2012; Negara et al. 2013), the empirical study focused on the following 9 within-document refactoring types: *extract class*, *extract method*, *extract variable*, *inline method*, *inline variable*, *rename attribute*, *rename method*, *rename parameter*, and *rename variable*. All such refactorings take a single Java document as input, and generate an improved version of it, which may simplify the prompt (and output) of LLMs. If either (or both) of the refactoring detection tools identified a refactoring of the

selected refactoring types, we collected it as a candidate and requested three refactoring experts (noted as PTC_A) to validate each potential refactoring independently and manually. In case of inconsistent validation, we simply discarded the case and turned to the next one. We collected the expert-validated refactorings into *ref-Dataset*. Notably, the process of analyzing the capability of LLMs to identify refactoring opportunities and evaluating the quality of LLMs' suggested solutions is tedious and time-consuming. To this end, we selected only one sample per refactoring type from each project, resulting in a total of 20 samples. This selection criterion was designed to maximize the diversity of the samples while minimizing the evaluation burden. If neither of the two tools identified a specific type of refactoring in a given project, we randomly selected another project from 20 selected projects where this type of refactoring had been discovered. After that, within that project, we selected the most recent expert-validated instance of the same refactoring type, following the chronological order of commits where the refactorings were discovered, and collected it into *ref-Dataset*.[1] As a result, *ref-Dataset* comprises 180 real-world refactorings, encompassing 9 distinct types of refactorings from 20 open-source projects. The dataset includes 160 unique Java documents and a single Java document may contain multiple refactorings. We also randomly selected one Java document from the latest version of each project (for a total of 20 Java documents). These documents were chosen as samples without any refactoring opportunities.

To validate the potential of LLM-based refactoring, we should feed source code containing refactoring opportunities and expect LLMs to generate the refactored version of the input. By discovering refactoring histories, we have V_n and V_{n-1} for each discovered refactoring in *ref-Dataset* where V_{n-1} represents the source code before refactoring and V_n represents the source code after refactoring. An intuitive idea is to feed V_{n-1} to LLMs, and expect them to generate V_n. However, it is impractical because of the following reasons. First, the commit changing V_{n-1} to V_n often contains other changes (e.g., functional changes and bug fixing) besides refactorings (Silva et al. 2016). Consequently, LLMs with V_{n-1} as input may not generate V_n by applying refactorings only. Second, refactoring operations are often motivated by other non-refactoring changes. For example, *rename method* refactoring is often conducted when developers are modifying the functionality of a method. A typical example is presented in Fig. 1. In this example, the developer removed the method call "*normalize*()" (Lines 264–265), and renamed the method from *relativizeAndNormalizePath* to *relativizePath* because the modified method body is inconsistent with the original method name. Consequently, without such motivating non-refactoring changes (within the same commit), the refactorings may become unnecessary and thus LLMs cannot report such refactoring opportunities.

To this end, we created suitable input (request) to LLMs with reverse refactoring as shown in Fig. 2. A reverse refactoring is a refactoring that removes an existing refactoring. For example, if a refactoring renames a method from *oldName* to *newName*, its corresponding reverse refactoring is to rename the same

[1] https://github.com/bitselab/LLM4Refactoring/blob/master/data/dataset.

```
258     -   public static String relativizeAndNormalizePath(final String baseDirectory,
                                                           final String path) {
        258 +   public static String relativizePath(final String baseDirectory,
                                                    final String path) {
259 259         final String resultPath;
260 260         if (baseDirectory == null) {
261 261             resultPath = path;
262 262         }
263 263         else {
264     -           final Path Path pathAbsolute = Paths.get(path).normalize();
265     -           final Path pathBase = Paths.get(baseDirectory).normalize();
        264 +       final Path pathAbsolute = Paths.get(path);
        265 +       final Path pathBase = Paths.get(baseDirectory);
266 266           resultPath = pathBase.relativize(pathAbsolute).toString();
267 267         }
268 268         return resultPath;
269 269     }
```

Fig. 1 An example of rename method refactorings from commit d52eb5d in checkstyle

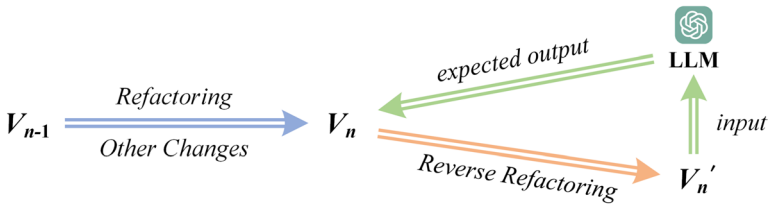


Fig. 2 Creating input and expected output of LLMs

method from *newName* to *oldName*. From the refactored version V_n, we manually applied reverse refactoring to remove the refactoring conducted by the given commit. As a result, V_n' has all of the changes made by the commit except for the discovered refactoring. Consequently, it has all of the motivation changes (if there are any) as well as the refactoring opportunity. In conclusion, we take V_n' as the input to the LLMs and V_n as the expected output. Notably, V_n' is equivalent to V_{n-1} with other changes, which are generally available to developers.

2.3 Prompt template

The quality of prompts could significantly influence the performance of LLMs (White et al. 2023, 2024). To select suitable prompts, we followed well-established best practices (Akın 2024; Dair.AI 2024) which suggests that prompts could consist of *instruction*, *context*, *input data*, and *output indicator* for better results. Moreover, in a recent study, Guo et al. (2024) explored the potential of ChatGPT in automated code refinement and designed prompt templates that could maximize the performance of ChatGPT. Their findings suggest that prompts with concise scenario descriptions tend to generate better results. To this end, in this paper, we followed

the best practices and tailored different prompt templates to answer the different research questions (as shown in Fig. 3):

(1) To answer RQ1-1, we designed the first prompt template **P1**. It specifies the role of the large language model, the task, and the source code to be refactored.
(2) To answer RQ1-2, we designed the second prompt template **P2**. Besides all the information specified in P1, P2 also specifies the requested refactoring type. Notably, we designed a unique version of P2 for each refactoring type.
(3) To answer RQ2, we designed the third prompt template **P3**. It specifies the role of the large language model, the task, the code entities that should be refactored, and the requested refactoring type as well as its explanation.

Notably, the *original code* in the templates refers to the document (file) where the refactoring is expected to be done. As we explained in Sect. 2.2, all of the selected refactorings are within-document refactoring whose key modifications are often limited to a single document. Although source code outside the enclosing document may influence the refactoring and some modifications could happen to other documents, it is impractical to input all such contexts into the prompt because of the

Prompt 1 (P1):

As a developer, imagine your team leader requests you to review a piece of code to identify potential refactoring opportunities. The original code snippet is as follows:
``` *original code* ```
If there are any refactoring opportunities, please do it and generate the refactored code. Otherwise, simply state that no refactoring is necessary.

Prompt 2 (P2):

Rename method refactorings are frequently employed to modify low-quality identifiers to improve readability. As a developer, imagine your team leader requests you to review a piece of code to identify potential rename method refactoring opportunities. The original code snippet is as follows:
``` *original code* ```
If there are any refactoring opportunities, please do it and generate the refactored code. Otherwise, simply state that no refactoring is necessary.

Prompt 3 (P3):

As a developer, imagine your team leader requests you to refactor a piece of code. The original code snippet is as follows:
``` *original code* ```
Rename method refactorings are frequently employed to modify low-quality identifiers to improve readability. In the preceding code, there is a low-quality method name:
``` *low-quality name* ```
You may employ rename method refactorings to improve the preceding code and generate the refactored code.

Fig. 3 Prompt templates

length limitation of the LLMs. For example, a *rename method* refactoring could be influenced by references (including those outside the enclosing document) to the method, and it should also update such references accordingly. However, its references could appear in many documents, and it is challenging to include all these documents in the prompt.

2.4 Metrics

To evaluate the accuracy of LLMs in identifying refactoring opportunities, we define specific heuristics for each refactoring type. For a refactoring applied to a single code entity, like *rename method* and *inline variable* refactorings, if the conducted refactoring and the suggested refactoring are applied to the same entity (e.g., renaming the same method or inlining the same variable), we say the model has identified the refactoring opportunity. For a refactoring that involves a range of code entities, like *extract method* and *extract class* refactorings, if the suggested refactoring is applied to the same enclosing entity (i.e., the decomposed long method for extract method refactoring, and the decomposed large class for extract class refactoring) and they modify some common lines of source code (e.g., they extract some common statements or common code entities), we say the model succeeds in identifying the refactoring opportunity. Specifically, we employ well-known and widely-used metric *dice coefficient* (Dice 1945; Fluri et al. 2007; Falleri et al. 2014), defined as:

$$tolerance = \frac{2 \times \#commons}{\#extracted + \#oracle} \tag{1}$$

, where *#commons* indicates the number of statements (code entities) extracted by LLMs are common with the refactoring oracle, to evaluate whether LLMs identify *extract method* and *extract class* refactoring opportunities. If the *tolerance* is greater than or equal to 0.5, i.e., more than half of the extracted statements (code entities) align with the refactoring oracle's suggestions, we consider that the model successfully identifies the refactoring opportunity. Otherwise, the model is deemed to fail in identifying the refactoring opportunity.

To evaluate the quality of refactorings conducted by LLMs, we requested three experienced developers (noted as PTC_B) to assess the quality of the refactorings conducted by LLMs. Note that PTC_B had no overlap with the participants in Sect. 2.2. All of the participants in PTC_B had rich Java experience and were familiar with software refactoring. They had a median of 10 years of programming experience, and 4.5 years working as professional software developers. Given the original code and the refactored version (i.e., the output of LLMs), each participant should depict the quality as one of the five categories:

- Excellent: better than developers, score = 4;
- Good: comparable (or identical) to developers, score = 3;
- Poor: inferior to developers, score = 2;
- Failed: no refactoring at all, score = 1;

- Buggy: resulting in semantic or syntactic bugs, score = 0.

Notably, if the solution is buggy (i.e., changing the functionality of the source code or resulting in compilation errors), we requested the participants to mark it as *buggy* even if the solution could be excellent (or good) when the bug is fixed. All participants rated independently, and they reached a high consistency by achieving a Fleiss' kappa coefficient (Fleiss 1971) of 0.82. For each solution, if any of the three participants rated it as *buggy*, we requested them to discuss it together and reach a consensus. We took the median of the three ratings (of the same refactoring solution) as its final rating.

3 Identification of refactoring opportunities

3.1 RQ1-1: identification with generic commands

To answer RQ1-1, we requested GPT and Gemini to refactor the given original code using prompt template P1 introduced in Sect. 2.3 as a cue. The evaluation results are presented in Table 1 where #TPs, #FPs, and #FNs represent the number of successfully identified refactoring opportunities, the number of incorrectly identified refactoring opportunities, and the number of missed refactoring opportunities, respectively. The percentages within parentheses present the Recall = #TP/(#TP+#FN) and Precsion = #TP/(#TP+#FP), where Recall measures the ratio of the number of succeeded cases to the total number of involved cases and Precision measures the ratio of the number of correct cases to the total number of identified cases.

From Table 1, we observed that GPT and Gemini successfully identified 28 and 7 out of the 180 refactoring opportunities, respectively. As a result, their success rates (i.e., Recall) were 15.6% = 28/180 and 3.9% = 7/180, respectively. That is, they missed many more refactoring opportunities than what they successfully identified. It suggests that LLM-based identification of refactoring opportunities with generic commands may not be fruitful. We also observed that GPT and Gemini resulted in a significant number of false positives (i.e., refactorings that were not actually conducted by the original developers), with a precision of only 23.5% and 10% respectively.

We further analyzed the evaluation results by considering the performance on different refactoring types (as shown in Table 1). Our analysis results suggest that:

- Neither of them identified any of the 20 extract class refactoring opportunities, suggesting that it is rather challenging for LLMs to identify this kind of refactoring opportunities.
- GPT achieved a success rate of 45% in identifying extract method refactoring opportunities. It is also the highest success rate on the table. However, we also notice that the success rate of Gemini on the same refactoring type is as low as 5%. It may suggest that not all LLMs are good at identifying extract method refactoring opportunities.

Table 1 LLM-based identification of refactoring opportunities

Refactoring type	GPT			Gemini		
	#TPs (R)	#FPs (P)	#FNs	#TPs (R)	#FPs (P)	#FNs
Extract class	0 (0%)	3 (0%)	20	0 (0%)	4 (0%)	20
Extract method	9 (45%)	41 (18%)	11	1 (5%)	27 (3.6%)	19
Extract variable	5 (25%)	5 (50%)	15	0 (0%)	5 (0%)	20
Inline method	3 (15%)	2 (60%)	17	2 (10%)	1 (66.7%)	18
Inline variable	3 (15%)	9 (25%)	17	1 (5%)	2 (33.3%)	19
Rename attribute	1 (5%)	4 (20%)	19	0 (0%)	5 (0%)	20
Rename method	2 (10%)	14 (12.5%)	18	0 (0%)	11 (0%)	20
Rename parameter	3 (15%)	8 (27.3%)	17	0 (0%)	4 (0%)	20
Rename variable	2 (10%)	5 (28.6%)	18	2 (10%)	4 (33.3%)	18
Total	28 (15.6%)	91 (23.5%)	152	7 (3.9%)	63 (10%)	173

- Gemini failed to outperform GPT on any of the refactoring types. On six out of the nine refactoring types, GPT substantially outperformed Gemini. On the other three refactoring types (i.e., extract class, inline method, and rename variable), they fought to a draw. As a result, GPT identified 28 refactoring opportunities in total, 4 = 28/7 times the number identified by Gemini.
- Both GPT and Gemini resulted in the highest number of false positives in the extract method, while producing the fewest false positives in the inline method. These findings suggest that LLMs may have a tendency to break down methods into smaller ones rather than inlining methods into their callers.
- GPT identified more refactoring opportunities than Gemini across 6 out of the 9 refactoring types, leading to a higher number of false positives.

To reveal what kind of changes LLMs had conducted and why they frequently failed to identify the given refactoring opportunities, we retrieved three examples for each of the nine refactoring types from the refactoring results of GPT and Gemini, resulting in a total of 54 examples (2 LLMs × 9 refactoring types × 3 examples). These examples were retrieved randomly, and the selected documents included both cases where LLMs succeeded in identifying refactoring opportunities and failing cases. We manually review the changes made by the LLMs and validate how many of them correspond to actual refactoring operations, as identified refactoring opportunities. We then evaluate whether the refactoring opportunities identified by the LLMs contribute to improved code quality. If they do, the refactoring is classified as beneficial; Otherwise, it is deemed unhelpful. Our evaluation results suggest that from the 54 sampled examples, LLMs identified a total of 112 refactoring opportunities, including 80 opportunities of the nine refactoring types. 76.8% = 86/112 (i.e., accuracy) of the suggested refactoring opportunities were manually confirmed as beneficial whereas others were denied. Further analysis on the 80 refactoring opportunities suggests that:

- 72.5% = 58/80 of them were associated with extract-related refactorings (especially extract method refactorings), 22.5% = 18/80 were associated with rename-related refactorings, and 5% = 4/80 were associated with inline-related refactorings. The distribution also explains why LLMs frequently failed to identify the given types of refactoring opportunities except for extract method refactoring opportunities.
- 60.3% = 35/58 of the extract-related suggestions were employed to alleviate duplicate code and the remainder focused on decomposing large classes/methods or extracting expressions (code snippets). In total, 72.4% = 42/58 of such refactoring opportunities were manually confirmed as beneficial whereas others were denied.
- 72.2% = 13/18 of the rename-related refactorings suggested employing more descriptive names whereas the others suggested maintaining consistent naming styles by renaming. In total, 61.1% = 11/18 of such refactoring opportunities were manually confirmed as beneficial whereas others were denied.

Besides the nine refactoring types, LLMs also suggested 32 refactoring opportunities of other types, e.g., *simplifying code structures* and *leveraging Java's new features*. For example, LLMs suggested replacing loop statements with stream API to improve code readability and running efficiency. Notably, such kinds of refactorings are less popular than the nine refactoring types investigated in this paper and some of them may not yet supported by mainstream refactoring tools, which may indicate that LLMs have the potential to suggest refactoring opportunities of less famous refactoring types. In total, 25 out of the 32 refactoring opportunities were manually confirmed as beneficial, with an accuracy of 78.1% = 25/32.

Answer to RQ1-1 LLMs exhibit limited effectiveness in identifying refactoring opportunities with generic prompts, with a success rate of only 15.6%. Moreover, LLMs tend to produce a significant number of false positives, with a precision of only 23.5%.

3.2 RQ1-2: explicitly specifying expected refactoring types

To investigate the performance of LLMs in identifying refactoring opportunities when expected refactoring types are explicitly specified, we requested GPT and Gemini to refactor the given original code using prompt template P2 as a cue. For each refactored version of the LLMs, we manually analyzed whether the LLMs actually performed refactoring and whether the type of refactoring aligned with the ground truth. In addition, for *extract class* and *extract method* refactorings, we employed the tolerance criteria outlined in Sect. 2.4 to assess whether the LLMs successfully identified potential refactoring opportunities. Consequently, we also manually analyzed the overlap between the statements (code entities) extracted by LLMs and the ground truth. Based on the manual analysis, we computed the performance metrics accordingly.

The evaluation results are presented in Table 2. By comparing Tables 1 and 2, we observe that explicitly specifying the expected refactoring types can substantially improve the success rate in LLM-based identification of refactoring opportunities. The success rate of GPT increased substantially from 15.6 to 52.2%, with a relative improvement of 234.6% = (52.2% − 15.6%)/15.6%. Gemini also achieved a substantial improvement of 441% = (21.1% − 3.9%)/3.9%. We performed the Wilcoxon signed-rank test (Wilcoxon 1945) and used Cliff's Delta (d) as the effect size (Grissom and Kim 2005) to validate whether explicitly specifying expected refactoring types significantly improved the success rate. The test results (p-value = 3.06E−15, Cliff's $|d| = 0.37$) confirmed that the improvement of GPT was statistically significant. Similarly, the improvement of Gemini was statistically significant (p-value = 1.61E−7 and Cliff's $|d| = 0.17$).

From Table 2, we also observe that the success rate varies substantially among different refactoring types:

- Explicitly specifying the expected refactoring types did not reduce LLMs' success rate on any refactoring types.
- Explicitly specifying the expected refactoring types improved the success rate of GPT on all types of refactoring opportunities whereas it only improved Gemini on four out of the nine categories of refactoring opportunities, i.e., extract method, rename attribute, rename method, and rename parameter.
- The success rate of GPT varied from 25 to 70% whereas the success rate of Gemini varied from 5 to 40%. It may suggested that LLM-based refactoring recommendation could be more fruitful on some refactoring types.
- GPT achieved a high success rate of 67.5% = (54/80) in identifying rename refactoring opportunities, including rename attribute, rename method, rename parameter, and rename variable. However, we also observed that Gemini's success rate on rename refactoring was not substantially higher than that on other refactorings. It may suggest that different large language models could be good

Table 2 LLM-based identification of refactoring opportunities (specifying expected refactoring types

Refactoring type	GPT			Gemini		
	#TPs (R)	#FPs (P)	#FNs	#TPs (R)	#FPs (P)	#FNs
Extract class	9 (45%)	29 (23.7%)	11	2 (10%)	37 (5.1%)	18
Extract method	12 (60%)	84 (12.5%)	8	5 (25%)	39 (11.4%)	15
Extract variable	7 (35%)	55 (11.3%)	13	1 (5%)	33 (2.9%)	19
Inline method	7 (35%)	35 (16.7%)	13	6 (30%)	25 (19.4%)	14
Inline variable	5 (25%)	53 (8.6%)	15	3 (15%)	29 (9.4%)	17
Rename attribute	14 (70%)	92 (13.2%)	6	8 (40%)	42 (16%)	12
Rename method	12 (60%)	172 (6.5%)	8	6 (30%)	46 (11.5%)	14
Rename parameter	14 (70%)	127 (9.9%)	6	4 (20%)	36 (10%)	16
Rename variable	14 (70%)	87 (13.9%)	6	3 (15%)	19 (13.6%)	17
Total	94 (52.2%)	734 (11.4%)	86	38 (21.1%)	306 (11%)	142

at identifying different types of refactoring opportunities, and thus they are potentially complementary.
- The success rate on inline refactoring opportunities (i.e., inline method and inline variable) remained low even if the expected refactoring types were explicitly specified.
- When the expected refactoring types were explicitly specified, the false positives for GPT and Gemini increased by $706.6\% = (734 - 91)/91$ and $385.7\% = (306 - 63)/63$, respectively. These findings suggest that LLMs face challenges in accurately identifying genuine refactoring opportunities.
- Both GPT and Gemini exhibited the highest number of false positives when the rename method was explicitly specified. These findings suggest that LLMs tend to favor renaming methods when explicitly instructed to do so.

When the expected refactoring types are explicitly specified, we observe that LLMs significantly increase the number of identified refactoring opportunities. GPT achieved an improvement of $235.7\% = (94 - 28)/28$, while Gemini achieved an improvement of $442.9\% = (38 - 7)/7$. We leverage the example in Listing 1 to illustrate that LLMs can accurately identify refactoring opportunities that are consistent with those actually conducted by developers.

```
1    /* From DefaultRefreshEventListener.java in Hibernate-ORM */
2    public void onRefresh(RefreshEvent event, RefreshContext
      refreshedAlready) {
3        ......
4        if (persistenceContext.reassociateIfUninitializedProxy(object)) {
5  +         boolean isTransient = isTransient(event, source, object);
6             ......
7  -         if (isTransient(event, source, object)) {
8  +         if (isTransient) {
9                 source.setReadOnly(object, source.isDefaultReadOnly());
10            }
11       ......
12   }
```

Listing 1: Refactoring Opportunity Successfully Identified by LLM

In this example, the original developer extracted the conditional expression in the *if* statement (Line 7) into the variable (Line 5) to improve the readability and maintainability of the code. GPT also accurately identified this refactoring opportunity and conducted the same *extract variable* refactoring as conducted by the developer. Therefore, we unanimously believe that GPT successfully identified a refactoring opportunity (i.e., #Succeeded).

To compute the accuracy in identifying refactoring opportunities, we recruited three evaluators (i.e., PTC_B as introduced in Sect. 2.4) to manually validate all refactorings conducted by LLMs. If LLMs conducted a refactoring of the expected refactoring type and this refactoring was not found in the ground truth, it is a false positive if at least two of the three evaluators deemed it *unnecessary*. Note that a refactoring is unnecessary if it cannot improve the quality of the source code. The evaluation results suggest that both GPT and Gemini reported some false positives, which had a negative impact on their accuracy. For example, GPT suggested 828 refactorings of the expected types, of which 501 were considered necessary and 327 were denied by human experts. As a result, the accuracy of GPT is $60.5\% = 501/828$. Gemini achieved a comparable accuracy of $53.8\% = 185/344$.

We tried to add all expected refactoring types to the prompt and evaluated whether LLMs can accurately identify refactoring opportunities. The evaluation results are presented in Table 3. From this table, we conclude that the prompt with all refactoring types is more effective than the generic prompt but less effective than the prompt specifying expected refactoring types.

Answer to RQ1-2 When refactoring types are explicitly specified, the success rate of LLMs increases significantly from 15.6 to 52.2%. However, some refactoring types, especially *inline* refactorings, still have a low success rate.

3.3 RQ1-3: size of source code

To investigate how the size of the source code influences the success rate, we calculated the point-biserial correlation coefficients (Tate 1954) between the size of the to-be-refactored source code and the success rate of LLMs in identifying refactoring opportunities. We measured the size of the source code with the well-known metric LOC (lines of code).

Our computation results suggest that when prompt template P2 was used, there is a moderate (GPT, −0.38) or weak (Gemini, −0.28) negative correlation with the size of the to-be-refactored source code: The larger the source code is, the lower the success rate is. When we switched from prompt template P2 to P1 (without explicitly specifying the expected refactoring types), the negative correlation remained (−0.29 for GPT and −0.2 for Gemini). Besides the correlation coefficient, we categorized the source code into four equal-sized groups according to their size (LOC), noted as Q1, Q2, Q3, and Q4 where Q1 was composed of the 25% of the smallest code snippets. The evaluation results are presented in Fig. 4a. The horizontal axis outlines the four groups, along with the minimum and maximum sizes of LOC within the given

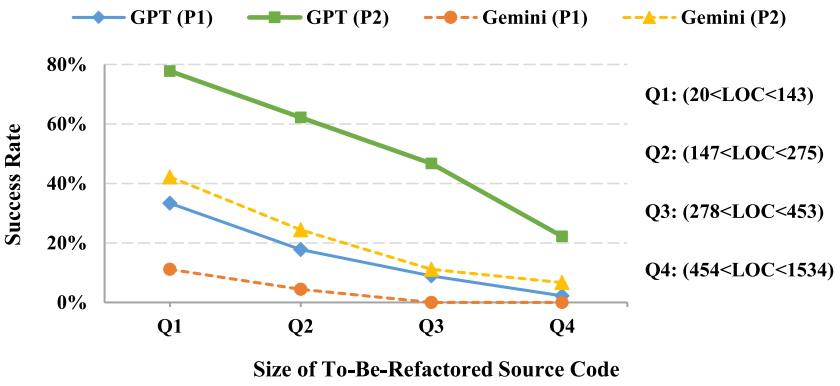
Table 3 LLM-based identification of refactoring opportunities (with all refactoring types

Refactoring type	GPT			Gemini		
	#TPs (R)	#FPs (P)	#FNs	#TPs (R)	#FPs (P)	#FNs
Extract class	3 (15%)	13 (18.8%)	17	0 (0%)	19 (0%)	20
Extract method	8 (40%)	47 (14.5%)	12	1 (5%)	22 (4.3%)	19
Extract variable	2 (10%)	26 (7.1%)	18	0 (0%)	20 (0%)	20
Inline method	4 (20%)	17 (19%)	16	2 (10%)	18 (10%)	18
Inline variable	7 (35%)	9 (43.8%)	13	1 (15%)	20 (4.8%)	19
Rename attribute	9 (45%)	19 (32.1%)	11	2 (10%)	16 (11.1%)	18
Rename method	3 (15%)	38 (7.3%)	17	1 (5%)	24 (4%)	19
Rename parameter	5 (25%)	13 (27.8%)	15	2 (10%)	20 (9.1%)	18
Rename variable	8 (40%)	21 (27.6%)	12	2 (10%)	10 (16.7%)	18
Total	49 (27.2%)	203 (19.4%)	131	11 (6.1%)	169 (6.1%)	169

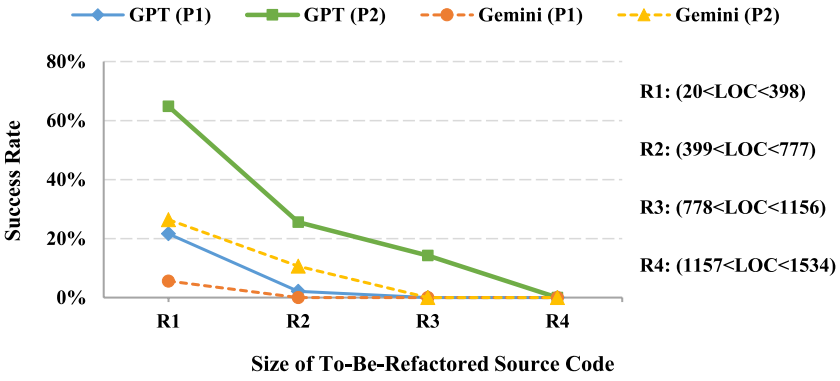
group. The vertical axis represents the success rate in refactoring opportunity identification. We observe from this figure that the success rate decreased quickly with the increase in LOC. We also categorized the source code into four groups based on the range of lines of code (LOC), noted as R1, R2, R3, and R4, where each range spanned 378 LOC. The evaluation results are presented in Fig. 4b.

The finding, i.e., it is more challenging to identify the given refactoring opportunities accurately from longer to-be-refactoring source code, is reasonable. It is well-known that the longer the prompt is, the more challenging it is for LLMs to understand the input and generate expected answers (Chang and Fosler-Lussier 2023). Consequently, when the to-be-refactored source code is lengthy and complex, it is reasonable that LLM-based identification of refactoring opportunities could be less successful. Notably, the evaluation results suggest that for Q3 and Q4 (i.e., LOC > 278), the success rate of LLMs diminishes quickly. We conclude from the results that the practicality of LLMs in identifying refactoring is confined to small code fragments of less than 300 LOC. We also observe that GPT identified more refactoring opportunities than Gemini under the same prompt. Furthermore, Gemini demonstrates a higher success rate in identifying refactoring opportunities when the proper prompts are given (i.e., P2), compared to GPT when the refactoring types are not explicitly specified (i.e., P1). The finding highlights the importance of prompt settings in maximizing the potential of LLMs in identifying refactoring opportunities.

Answer to RQ1-3 The size of the source code negatively influences the success rate of LLMs in identifying refactoring opportunities, with longer source code resulting in lower success rates. Furthermore, LLMs demonstrate limited practicality when handling source code that exceeds 300 LOC.



(a) Equal-sized Groups



(b) Equal-ranged Groups

Fig. 4 Size of source code influences LLMs' success rate

3.4 RQ1-4: strengths and weakness of LLMs

To gain a deeper understanding of the strengths and weaknesses of LLM-based identification of refactoring opportunities, we manually analyzed the reasons behind the refactorings in the subject dataset and classified them into 23 refactoring subcategories (reasons). For example, *code duplication*, one of the most common code smells, refers to the duplicated code snippets within a software project (Fowler 1999; Yamashita and Moonen 2013). The *extract method* refactoring is an effective and commonly employed strategy to mitigate this issue (Silva et al. 2016; AlOmar et al. 2024). Figure 5 illustrates the taxonomy of these subcategories. The number on the top-right corner of each label in this figure represents the number of instances manually assigned to each refactoring subcategory (e.g., 14 refers to *inconsistent method name*). In the previous research questions (RQs), we observe that GPT consistently outperformed Gemini across all types of refactorings and in both prompt settings. Consequently, this section concentrates on cases where the state-of-the-art

LLM (i.e., GPT) performed well or not, and explores potential improvements in its capability to identify refactoring opportunities.

The blue bar represents the success rate of GPT in identifying refactoring opportunities under P2. An empty bar represents that GPT always failed in that category. From Fig. 5, we make the following observations:

- GPT performed poorly in identifying inline-related refactoring opportunities, especially for variables that are only references to other objects (*use as reference*) and methods that are only called by a proxy method (*proxy method*). In both cases, GPT failed to identify any refactoring opportunities, suggesting that GPT does not excel at identifying such refactoring opportunities.
- For extract-related refactorings, the success rates did not significantly vary across subcategories. One possible explanation is that GPT possesses a robust understanding of the motives categorized for these refactorings. We also observe that GPT showed greater proficiency in removing code duplication compared to decomposing classes and methods as well as extracting complex expressions.
- Although the reasons for rename-related refactoring are more diverse than the other two types of refactorings, GPT showed the highest success rate in identifying such refactoring opportunities. However, the effectiveness of GPT varied for refactoring subcategories. For example, it rarely identified refactoring opportuni-

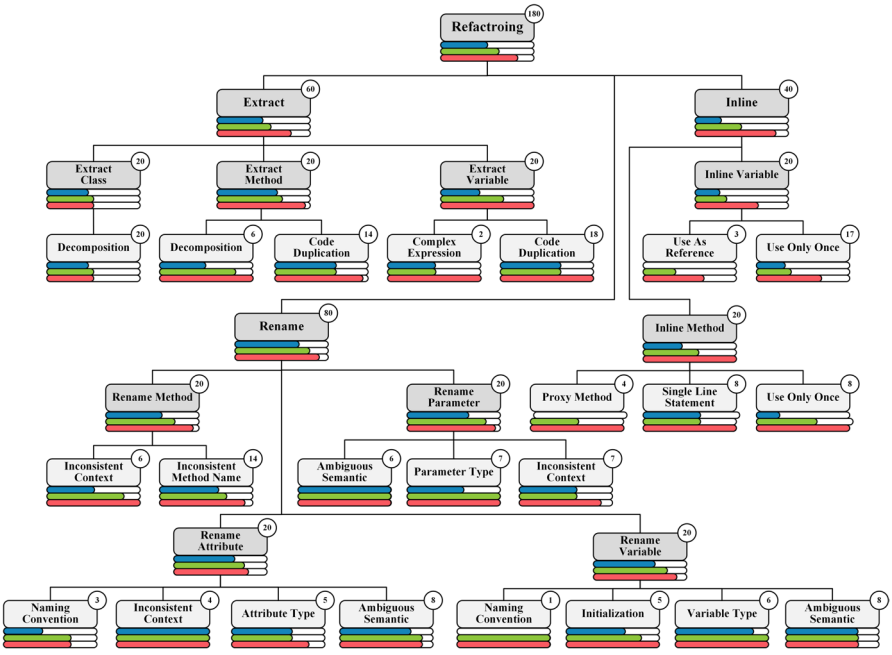


Fig. 5 Taxonomy of refactoring subcategories for automated software refactoring. We present the percentage of successful identification for each refactoring subcategory with bars below the corresponding category: P2 (blue bar), P2 + Refactoring Subcategory (green bar), P2 + Refactoring Subcategory + Search Space Limitation (red bar) (Color figure online)

ties associated with *naming conventions*, whereas it had a higher success rate in identifying refactoring opportunities for *ambiguous semantics*.

To further investigate the performance of GPT in identifying refactoring opportunities across different subcategories (e.g., *code duplication* and *naming convention*), we designed a new prompt template **P2 + Refactoring Subcategory** for each refactoring subcategory, as illustrated in Fig. 6.

With the increasing ability of LLMs, in-context learning has become more widely adopted. By providing a few examples of desired inputs and outputs within the context, LLMs can better understand and adapt to specific tasks (e.g., refactoring opportunity identification), enhancing their effectiveness. To this end, we retrieve similar refactoring patterns from the history of the subject projects to serve as contextual cues, instructing the LLMs to understand the necessary background knowledge for each refactoring subcategory. Notably, the retrieved examples can be used repeatedly because the refactoring patterns corresponding to each refactoring subcategory are consistent. For example, expressions that are repeatly used should be extrated as a new variable to enhance code readability and maintainability.

The full prompt templates are available in our replication package (Liu 2024b). As presented in Fig. 5, the green bar represents the success rate of GPT in identifying refactoring opportunities under P2 + Refactoring Subcategory. We observe from this figure that explicitly specifying the expected refactoring Subcategories can improve the success rate in LLM-based identification of refactoring opportunities. Specifically, the success rate of GPT increased significantly from 52.2 to 66.7%, with a relative improvement of $27.8\% = (66.7\% - 52.2\%)/52.2\%$. Furthermore, there was no observed decrease in success rate across any subcategory when refactoring subcategories were explicitly specified. We performed the same significance test as described in Sect. 3.2 to validate whether explicitly specifying expected refactoring Subcategories significantly improved the success rate. The test results (p-value = 1.26E−4, Cliff's $|d| = 0.14$) confirmed that the improvement was statistically significant.

We conclude in Sect. 3.3 that the success rate of identifying refactoring opportunities is affected by the size of the source code. The finding highlights the challenge of LLMs in effectively understanding and handling longer and more complex source code. Consequently, it may not be a wise choice to feed the entire document to LLMs for specific refactoring types. For example, *extract variable* refactorings typically do not involve code snippets beyond the scope of the enclosing method of the extracted expressions. To this end, we tried to narrow the search space based on P2 + Refactoring Subcategory. Notably, for each refactoring subcategory, we designed a unique strategy to narrow the search space, noted as **Search Space Limitation**. In the example illustrated in Fig. 6a, the search space was limited from the entire document to a single method because the method name is only associated with the functionality of its method body. For complex refactorings, like *extract method*, the search space was also limited from the entire document to a single method, as illustrated in Fig. 6b. Notably, for *extract class* refactoring, the search space remains unchanged, as the class serves as the smallest unit for this type of refactoring. The full strategies are detailed in our replication package (Liu 2024b). As presented in Fig. 5, the red bar

Prompt 2 (P2) + Refactoring Subcategory:

Rename method refactorings are frequently employed to modify low-quality identifiers to improve readability.
###
A rename method refactoring pattern is to rename methods whose names are inconsistent with their bodies. Here is an example of the rename method refactoring that follows this pattern.
The source code before refactoring is:
``` *example code before refactoring* ```
The source code after refactoring is:
``` *example code after refactoring* ```
In this example, the developer renamed the method "***oldName***" to "***newName***".
###
As a developer, imagine your team leader requests you to review a piece of code to identify potential rename method refactoring opportunities. The original code snippet is as follows:
``` *original method* ```
If there are any refactoring opportunities, please do it and generate the refactored code. Otherwise, simply state that no refactoring is necessary.

(a) Rename Method + Inconsistent Method Name

Prompt 2 (P2) + Refactoring Subcategory:

Extract method refactorings are frequently employed to decompose complex methods to improve readability.
###
An extract method refactoring pattern is to extract a piece of code as a new method to decompose this method. Here is an example of the extract method refactoring that follows this pattern.
The source code before refactoring is:
``` *example code before refactoring* ```
The source code after refactoring is:
``` *example code after refactoring* ```
In this example, the developer extracted the following statements:
``` *statement*-1; *statement*-2; ⋯; *statements-n* ```
as new method "***extractedMethodName***".
###
As a developer, imagine your team leader requests you to review a piece of code to identify potential extract method refactoring opportunities. The original code snippet is as follows:
``` *original method* ```
If there are any refactoring opportunities, please do it and generate the refactored code. Otherwise, simply state that no refactoring is necessary.

(b) Extract Method + Decomposition

Fig. 6 P2 + refactoring subcategory

represents the success rate of GPT in identifying refactoring opportunities under P2 + Refactoring Subcategory + Search Space Limitation. From this figure, we observe that narrowing the search space significantly boosts the success rate of GPT in identifying refactoring opportunities from 66.7 to 86.7%, with a relative improvement
```

of 30% = (86.7% − 66.7%)/66.7%. Moreover, the proposed strategy did not reduce the performance of GPT in any refactoring subcategory. The formulation of such prompts is practical, as it aligns with common development scenarios in real-world software engineering. Imagine the following two typical scenarios: First, developers often rely on metric-driven tools to ensure code quality, addressing issues such as high complexity or excessive length in methods. For example, tools like PMD can analyze the codebase, detect problematic methods, and generate alerts. These alerts help developers identify the scope and type of necessary refactoring. Once the problematic methods are identified, they can be fed into LLMs to pinpoint potential *extract method* opportunities, providing targeted guidance for improving code quality. Another common scenario arises when developers integrate externally sourced or auto-generated code fragments into their projects. These fragments might originate from online resources, code-sharing platforms, or even LLM-generated suggestions. The modified methods often retain their original names, which may not accurately align with the updated implementation or the purpose within the new context. Such misalignments can lead to confusion and hinder maintainability. To this end, developers can leverage LLMs to assess the appropriateness of method names and identify opportunities for *rename method* refactoring, improving clarity and consistency across the codebase.

Table 4 presents the performance of GPT in identifying nine types of refactoring opportunities refactoring subcategories are explicitly specified and the search space is narrowed. From this table, we make the following observations:

- For *extract variable* and *inline method* refactorings, GPT successfully identified all of the refactoring opportunities. Notably, under the original P2, the success rate of GPT in identifying these two types of refactoring opportunities was only 35%, which may reveal that the proposed two strategies: "Refactoring Subcategory" and "Search Space Limitation" are both effective.
- Similarly, the application of these strategies significantly improved the identification of refactoring opportunities for *extract method*, *rename method*, *rename parameter*, and *rename variable* refactorings.
- The proposed strategies sometimes did not yield significant improvements for *extract class* and *rename attribute* refactorings, which reveals that GPT's capability in identifying such refactorings may be limited.
- Although the proposed strategies significantly improved GPT's success rate in identifying *inline variable* refactoring opportunities, its performance still required further improvement. For *extract class* refactoring, the performance of GPT remained unsatisfactory. Therefore, we suggest that more efforts should be invested in the future on how to exploit LLMs to more accurately identify such refactoring opportunities.
- The proposed strategies significantly reduced GPT's false positives in all types of refactoring, as narrowing the search space directs the attention of LLMs toward the specific code entities that developers are working on.

We also performed the significance test to validate whether narrowing the search space significantly improved the success rate. The test results (p-value = 5.22E−9,

Table 4 LLM-based identification of refactoring opportunities (specifying expected refactoring subcategories ($) and narrowing the search space (†))

Refactoring type	P2			$P2^{\$}$			$P2^{\$\dagger}$		
	#TPs (R)	#FPs (P)	#FNs	#TPs (R)	#FPs (P)	#FNs	#TPs (R)	#FPs (P)	#FNs
Extract class	9 (45%)	29 (23.7%)	11	10 (50%)	12 (45.5%)	10	10 (50%)	12 (45.5%)	10
Extract method	12 (60%)	84 (12.5%)	8	14 (70%)	52 (21.2%)	6	19 (95%)	12 (61.3%)	1
Extract variable	7 (35%)	55 (11.3%)	13	13 (65%)	34 (27.7%)	7	20 (100%)	11 (64.5%)	0
Inline method	7 (35%)	35 (16.7%)	13	12 (35%)	6 (66.7%)	8	20 (100%)	0 (100%)	0
Inline variable	5 (25%)	53 (8.6%)	15	7 (25%)	35 (16.7%)	13	14 (70%)	21 (40%)	6
Rename attribute	14 (70%)	92 (13.2%)	6	16 (80%)	22 (42.1%)	4	17 (85%)	17 (50%)	3
Rename method	12 (60%)	172 (6.5%)	8	15 (75%)	76 (16.5%)	5	19 (95%)	61 (23.8%)	1
Rename parameter	14 (70%)	127 (9.9%)	6	17 (85%)	43 (28.3%)	3	19 (95%)	15 (55.9%)	1
Rename variable	14 (70%)	87 (13.9%)	6	16 (70%)	40 (28.6%)	4	18 (90%)	29 (38.3%)	2
Total	94 (52.2%)	734 (11.4%)	86	120 (66.7%)	320 (27.3%)	60	156 (86.7%)	178 (46.7%)	24

Cliff's $|d| = 0.2$) confirmed that the improvement was statistically significant. Although the proposed strategies improve the capability of LLMs to identify refactoring opportunities, some cases still pose challenges for accurate identification. We illustrate the possible reasons with the example in Listing 2, where the original developer renamed the attribute "*MAX_CAPACITY*" to "*MIN_CAPACITY*" to better align with the underlying code intent, as described in Farmmamba (2024). However, in this example, GPT always failed to identify this refactoring opportunity. Under the optimal prompt setting, GPT described "*MAX_CAPACITY*" as "*A clear constant name indicating the maximum capacity of the cache.*", which suggests that it believed the original name aptly expressed the attribute's functionality. The failure of the LLMs to identify this refactoring opportunity may stem from several reasons:

```
   /* From RetryCache.java in Hadoop */
   public class RetryCache {
       public static final Logger LOG = LoggerFactory.getLogger(
    RetryCache.class);
       private final RetryCacheMetrics retryCacheMetrics;
-      private static final int MAX_CAPACITY = 16;
+      private static final int MIN_CAPACITY = 16;
       ......
       public RetryCache(String cacheName, double percentage, long
    expirationTime)   {
            int capacity = LightWeightGSet.computeCapacity(percentage,
    cacheName);
-           capacity = Math.max(capacity, MAX_CAPACITY);
+           capacity = Math.max(capacity, MIN_CAPACITY);
            ......
        }
   }
```

Listing 2: Refactoring Opportunity Missed by LL• **Literal Interpretation:** The name "*MAX_CAPACITY*" straightforwardly suggests the maximum capacity. Without explicit context suggesting that it should represent the minimum capacity, LLMs may not deduce the need for renaming.

- **Complexity of Code Intent:** "*MAX_CAPACITY*" serves as a lower bound in capacity calculations and may exceed the LLMs' inferential capabilities, which involves understanding the role of functions `Math.max` in that specific context.
- **Limitation of Training Data:** If the LLMs' training data lacks similar contexts or if such refactoring patterns are rare, the models may not generalize well to the specific scenario, leading to difficulty in identifying such refactoring opportunities.
- **Lack of Domain-Specific Knowledge:** Despite their broad programming knowledge, LLMs may not possess detailed insights into certain best practices or naming conventions essential for identifying such refactoring opportunities.

Answer to RQ1-4 LLMs perform better in areas like code duplication removal and rename-related refactorings, but struggle with class decomposition. Furthermore,

specifying refactoring subcategories and narrowing the search space significantly improve their performance.

4 Recommendation of refactoring solutions

4.1 RQ2-1: quality of refactoring solutions

To investigate the quality of refactorings suggested (and conducted) by LLMs, we requested them to refactor the given original code using P3 as a cue. As a result, GPT and Gemini recommended 176 and 137 refactoring solutions for the 180 requests. We established a test suite to evaluate the safety of refactoring solutions suggested by LLMs. The suite comprises a total of 102 unit tests, derived from two sources: 59 are from the inherent unit tests in the subject dataset, while the remaining 43 are generated for the original code (i.e., the entire document) using an automated unit test generation tool (Yaron 2024). Notably, the tool was selected because (1) it supports various Java versions and (2) it can generate unit tests directly on the source code.

We initially ran these unit tests on the original code before refactoring, and the results confirmed that all unit tests passed successfully. Subsequently, we executed them again on the refactoring solutions suggested by GPT and Gemini, respectively, resulting in a total of 4 refactoring solutions that failed to pass the unit tests. To ensure no bugs were overlooked, we also requested developers (i.e., PTC_B) to manually inspect both the refactoring solutions that passed the unit tests and those for which unit tests could not be generated. This dual approach, combining automated testing with manual inspection, is designed to provide a thorough evaluation of the safety of LLM-based refactoring. In addition, the developers were requested to rate the quality of the refactoring solutions. The qualitative assessment followed the 5-point Likert Scale (Dawes 2008). "Excellent" refers to solutions suggested by LLMs that outperform those constructed by original developers. "Good" refers to solutions suggested by LLMs that are comparable to or identical to those constructed by original developers. "Poor" refers to solutions suggested by LLMs that are inferior to those constructed by original developers. "Failed" refers to cases where the LLMs do not perform any refactoring. "Buggy" refers to solutions suggested by LLMs that introduce semantic or syntactic bugs.

Figure 7 presents the results of human evaluation of the suggested solutions. The numbers before parentheses are the absolute numbers of solutions receiving the given rating whereas the percentages within parentheses present the ratio of the absolute numbers to the total number of requests (i.e., 180). From this figure, we make the following observations:

- First, LLMs have the potential in automated suggestions of high-quality refactoring solutions. For GPT, 63.6% = (15+97)/176 of the suggested solutions are comparable to (even better than) those constructed by original developers who actually conducted the refactorings. Although Gemini is less accurate, more than half ($56.2\% = (5 + 72)/137$) of its solutions are comparable to or better than

manually constructed solutions. We also notice that refactoring solutions suggested by LLMs are sometimes identical to the ground truth (constructed by the original developers). Out of the 176 solutions suggested by GPT and the 137 solutions suggested by Gemini, 48 and 25 solutions respectively are identical to their corresponding ground truth.

- Second, refactoring solutions suggested by LLMs cannot be accepted with further analysis. We notice that around one-third of the suggested refactoring solutions ($29\% = 51/176$ for GPT and $37.2\% = 51/137$ for Gemini) are poor, i.e., inferior to the corresponding manually constructed refactoring solutions.
- Third, LLMs may fail to suggest any solutions for the given refactorings. GPT and Gemini failed to make suggestions on 4 and 43 out of the 180 requests.
- Finally, the refactoring solutions suggested by LLMs could be unsafe, i.e., buggy. $7.4\% = 13/176$ and $6.6\% = 9/137$ of the refactoring solutions suggested by GPT and Gemini respectively are buggy. Among such $22 = 13 + 9$ buggy solutions, 18 results in semantic changes and 4 results in syntax errors.

On some cases, the overall quality of the suggested refactoring solutions is acceptable although they may result in semantic and/or syntactic bugs. That is, if the bugs are fixed, the solutions are comparable or even better than manually constructed ones. To validate this assumption, we manually fixed bugs and requested the participants to assess the refactorings again. Our evaluation results suggest that such manually fixed refactorings were frequently marked as good (14 out of 22 cases), with 2 marked as excellent and 6 marked as poor.

We further investigate how well the LLMs in suggesting solutions for different refactoring types, and the evaluation results are presented in Table 5. From this table, we make the following observations:

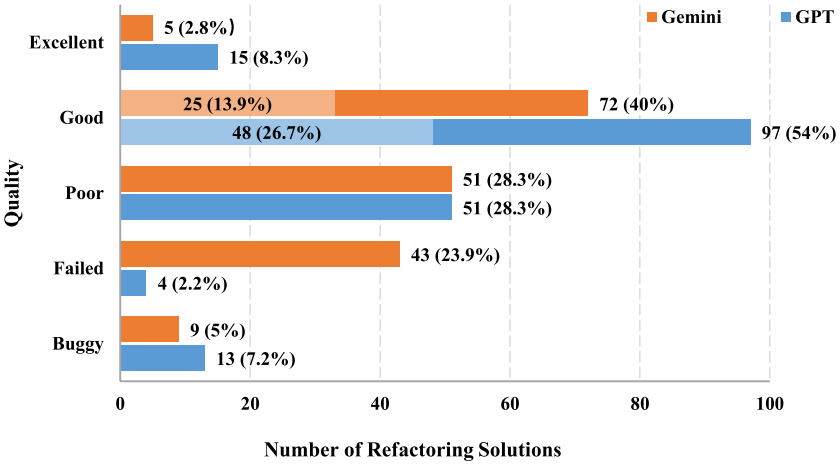


Fig. 7 Quality of LLMs' refactoring solutions

- The quality of the suggested refactoring solutions varies substantially among different refactoring types. For example, the percentage of excellent GPT solutions varies from zero to 20%. The percentage of poor Gemini solutions also varies substantially from zero to 65%. It may suggest that it is often not equally challenging to recommend solutions for different types of refactorings.
- LLMs works well in recommending solutions for inline-related refactorings (i.e., *inline method* and *inline variable*) and extraction-related refactorings (i.e., *extract class*, *extract method*, and *extract variable*). We observe that 87.5% of the solutions to inline-related refactorings are comparable to or even better than those conducted by developers, and 70% of the solutions to extract-related refactorings are comparable to or even better than manually constructed ones.
- Suggesting new names for rename refactorings is often more challenging. We notice that only 43.8% of the refactoring solutions (i.e., new names for the to-be-renamed code entities) are comparable to or even better than manually constructed solutions.
- Finally, LLMs result in buggy solutions on all refactoring types except for rename variable.

Table 5 Quality of LLMs' refactoring solutions

Refactoring type	Model	Quality				
		#Excellent	#Good	#Poor	#Failed	#Buggy
Extract class	GPT	0 (0%)	13 (65%)	3 (15%)	3 (15%)	1 (5%)
	Gemini	0 (0%)	10 (50%)	5 (25%)	4 (20%)	1 (5%)
Extract method	GPT	3 (15%)	10 (50%)	4 (20%)	0 (0%)	3 (15%)
	Gemini	1 (5%)	9 (45%)	2 (10%)	8 (40%)	0 (0%)
Extract variable	GPT	4 (20%)	12 (60%)	1 (5%)	0 (0%)	3 (15%)
	Gemini	0 (0%)	8 (40%)	4 (20%)	6 (30%)	2 (10%)
Inline method	GPT	1 (5%)	18 (90%)	1 (5%)	0 (0%)	0 (0%)
	Gemini	0 (0%)	12 (60%)	1 (5%)	5 (25%)	2 (10%)
Inline variable	GPT	0 (0%)	16 (80%)	1 (5%)	0 (0%)	3 (15%)
	Gemini	0 (0%)	13 (65%)	0 (0%)	5 (25%)	2 (10%)
Rename attribute	GPT	1 (5%)	6 (30%)	11 (55%)	1 (5%)	1 (5%)
	Gemini	0 (0%)	7 (35%)	9 (45%)	3 (15%)	1 (5%)
Rename method	GPT	2 (10%)	5 (25%)	12 (60%)	0 (0%)	1 (5%)
	Gemini	1 (5%)	4 (20%)	13 (65%)	2(10%)	0 (0%)
Rename parameter	GPT	2 (10%)	6 (30%)	11 (55%)	0 (0%)	1 (5%)
	Gemini	1 (5%)	4 (20%)	8 (40%)	6 (30%)	1 (5%)
Rename variable	GPT	2 (10%)	11 (55%)	7 (35%)	0 (0%)	0 (0%)
	Gemini	2 (10%)	5 (25%)	9 (45%)	4(20%)	0 (0%)
Total	GPT	15 (8.3%)	97 (54%)	51 (28.3%)	4 (2.2%)	13 (7.2%)
	Gemini	5 (2.8%)	72 (40%)	51 (28.3%)	43 (23.9%)	9 (5%)

```
1     /* From XmlPrinterTest.java in JavaParser */
2     private void assertXMLEquals(
3   -               String xml1, String actual) { // original parameter
4   +               String expected, String actual) { // ground truth
5   +               String expectedXML, String actualXML) { // GPT' result
6           final Document expectedDocument = getDocument(xml1);
7   +                                         getDocument(expected);
8   +                                         getDocument(expectedXML);
9           final Document actualDocument = getDocument(actual);
10  +                                       getDocument(actual);
11  +                                       getDocument(actualXML);
12          ......
13    }
```

Listing 3: LLM's suggested solution outperforms developer's suggestion

We observe that many of the refactoring solutions conducted by LLMs tend to outperform developers' solutions. An example is presented in Listing 3. In this example, the developer renamed the parameter "*xml*1" into "*expected*" (Lines 3–4) because the original parameter name lacked clarity and understandability within this method. GPT suggested the new name "*expectedXML*" is even better because it not only specifies the difference between the two parameters (expected vs. actual) but also specifies what it is (an XML document). The structure of the suggested name (*adjective*+*noun*) is also highly preferred by parameter names. We also notice that GPT suggested naming the other parameter "*actual*" into "*actualXML*".

Answer to RQ2-1 LLMs demonstrate a promising potential in suggesting high-quality refactoring solutions, with over 60% of solutions often comparable to or even better than those crafted by human experts.

4.2 RQ2-2: safety of suggested refactoring solutions

As mentioned in the preceding section, refactoring solutions suggested by the evaluated large language models could be unsafe (buggy). Our evaluation results suggest that 13 solutions suggested by GPT are unsafe and Gemini suggested 9 unsafe solutions as well. We also notice that the error rate varies slightly from 6.6% (Gemini) to 7.4% (GPT), suggesting that different LLMs have comparable error rates.

```
1     /* From CreateBranchCommand.java in JGit */
2     public Ref call() {
3         ......
4         try (RevWalk revWalk = new RevWalk(repo)) {
5   -         Ref refToCheck = repo.findRef(R_HEADS + name);
6   -         boolean exists = refToCheck != null;
7   +         boolean exists = repo.findRef(R_HEADS + name) != null;
8               ......
9   -         String startPointFullName = null;
10  -         if (startPoint != null) {
11  -             Ref baseRef = repo.findRef(startPoint);
12  -             if (baseRef != null)
13  -                 startPointFullName = baseRef.getName();
14  -         }
15  +         String startPointFullName = startPoint != null ? repo
                              .findRef(startPoint).getName() : null;
16        ......
17    }
```

Listing 4: Suggested Solution That Changes Source Code's Function

We manually analyzed the problems with the unsafe solutions. Our evaluation results suggest that in most cases (18 out of the 22 cases) the suggested solutions are unsafe because they change the functionality of the involved source code. We call such cases as *semantic bugs* because they change the semantics of the source code. A typical example is presented in Listing 4. In this example, GPT conducted an *inline variable* refactoring (Lines 5–7) correctly. It also made some additional changes: Replacing the *if* statements with a ternary operator expression (Lines 10–16). However, the changes as shown in Lines 10–16 are buggy. The expression "*repo.findRef(startPoint).getName*" may result in a null pointer exception, whereas the original code (and the code refactored by the developer) would not throw a null pointer exception because of the null pointer exception checking on Line 13. In total, GPT and Gemini suggested 10 and 6 refactoring solutions respectively that changed the functionality of the involved source code.

```
1     /* From Checker.java in Checkstyle */
2     private void processFiles(List<File> files) {
3         String fileName = null;
4         try {
5             fileName = file.getAbsolutePath();
6 +           final String filePath = file.getPath(); // extracted variable
7               ......
8         }
9         catch (Exception ex) {
10              ......
11 -           throw new CheckstyleException("......" + file.getPath(), ex);
12 +           throw new CheckstyleException("......" + filePath, ex);
13        }
14        catch (Error error) {
15              ......
16 -           throw new Error("......" + file.getPath(), error);
17 +           throw new Error("......" + filePath, error);
18        }
19    }
```

Listing 5: Suggested Solution That Introduces Syntax Errors

On the other four cases, the refactoring solutions suggested by LLMs were unsafe because they introduced syntax errors, making the resulting source code syntactically incorrect and thus uncompilable. A typical example is presented in Listing 5. In this example, GPT conducted an *extract variable* refactoring. However, the refactored code would result in a compilation error because the extracted variable "*filePath*" (Line 6) is defined inside the *try* block and it cannot be referenced in the *catch* blocks (Lines 12 and 17). The original developer extracted the variable before the *try* block, which did not result in any compilation errors. In total, GPT and Gemini suggested 3 and 1 such kind of unsafe solutions.

Compared against the syntax errors, changes in functionality could be more dangerous because they are often harder to identify. Syntax errors could be found by reliable compilers, and thus they have little chance to influence the final products delivered to end users. However, functional (semantic) changes cannot be identified by compilers. Regression testing has the potential to identify functional changes, but software projects in the wild rarely have sufficient regression unit tests. Consequently, such unintended changes have the chance to be delivered to end users, leading to serious losses.

Answer to RQ2-2 The refactoring solutions suggested by LLMs may be unsafe, i.e., they introduce semantic bugs that change the intended functionality and syntax errors that result in uncompilable code.

5 Discussion

5.1 Threats to validity

The first threat to external validity is that only a small number of refactoring instances were selected for the empirical study. However, it is challenging to substantially increase the number because it is time-consuming to manually validate the refactoring opportunities identified by LLMs and the refactoring solutions suggested by LLMs. It took a total of 42 man-days to manually check how many of the refactoring opportunities identified by the LLMs were aligned with the ground truth, as well as to assess the quality of the refactoring solutions suggested by the LLMs. To minimize the threat, we employed a reproducible process to select high-quality refactoring instances from 20 open-source projects in different domains. The second threat to external validity is that we only selected two large language models (i.e., GPT and Gemini) for our study. The conclusions drawn on them may not hold for other large language models. These two models were selected because they had been widely reported as the state of the art (Achiam et al. 2023; Bang et al. 2023; Team et al. 2023). The third threat to external validity is that the empirical study was confined to Java because the state-of-the-art refactoring detection tools do not support programming languages other than Java. Although Silva et al. proposed RefDiff (Silva and Valente 2017; Silva et al. 2020), a refactoring detection approach that supports multiple programming languages, its detection performance (i.e., precision and recall) and the range of supported refactoring types are significantly inferior to those of the tools selected in this paper (Tsantalis et al. 2022). Finally, we only investigated within-document refactorings because of the length limitation posed by LLMs on their input and output. The evaluation results reported in this paper may not hold on cross-document refactorings (e.g., *move class* and *move method* refactorings). Notably, the limitation of input length remains a significant challenge for LLMs. As these models continue to advance, they have the potential to handle longer and more complex inputs. Nowadays, one possible solution is to instruct the LLMs to store and remember the context information by feeding them the relevant source code incrementally. After that, we can use the tailored prompt template, along with the source code to be refactored, as input to guide LLMs in generating the corresponding refactored code.

The first threat to construct validity is that the manual construction of the refactoring dataset could be inaccurate. To conduct our study, we requested three refactoring experts to manually validate the refactorings reported by the selected refactoring detection tools, and their validation served as the ground truth for our study. However, manual validation is often subjective, and thus it could be inaccurate. To mitigate this threat, we requested the participants to validate all cases independently and discard cases where inconsistency was found. The second threat to construct validity is that the manual rating conducted in Sect. 4.1 could be inaccurate. Note that the assessment of code quality is often challenging and subjective. To minimize the threat, we invited multiple qualified participants

to rate refactoring solutions independently. They achieved a Fleiss' kappa coefficient (Fleiss 1971) of 0.82, indicating a high level of agreement among them. It may suggest that the resulting ratings could be reliable.

A threat to statistical validity is the inherent randomness of LLMs' predictions. Even when the same input and model parameters are used, the results from LLMs may vary due to their stochastic nature. This variability may lead to different outcomes in repeated experiments, making it challenging to achieve consistent results and potentially undermining the reliability of findings. As a result, the reproducibility of experiments involving LLMs may be compromised. To minimize this threat, we sampled 20 examples from different subject projects for each type of refactoring to reduce potential statistical bias.

5.2 Limitations

The first limitation is that although our empirical study in Sects. 3 and 4 confirms that LLMs have the potential to identify refactoring opportunities and to suggest high-quality refactoring solutions, the study does not quantitatively investigate to what extent LLMs can help identify more refactoring opportunities or to what extent LLMs can help reduce the cost of software refactoring. Notably, developers currently conduct refactorings manually or with the help of supporting tools like code smell detection tools (e.g., JDeodorant (Fokaefs et al. 2007; Tsantalis and Chatzigeorgiou 2011) and PMD (PMD 2024)), refactoring solution advisors (e.g., RefBERT (Liu et al. 2023) and EM-Assist (Pomian et al. 2024a)), and refactoring engines (e.g., IntelliJ IDEA). In this paper, we do not empirically compare LLM-based refactoring against such manual or semi-automated approaches. Consequently, it remains unclear to what extent developers can benefit from LLM-based refactoring concerning the effect (i.e., quality improvement caused by conducted refactorings) and the cost (e.g., human cost and computational expense) compared against the state of the art.

Another limitation is that the selected refactoring detection tools may miss commits with valid refactorings. To address this limitation, we followed state-of-the-art procedures (Fontana et al. 2012; Fulop et al. 2008; Kniesel and Binun 2009) that used triangulation between multiple sources (refactoring detection tools and human experts) to discover and validate refactorings in commit history. First, we independently applied two state-of-the-art refactoring detection tools to discover refactorings. The selected tools used complementary heuristic rules, and thus they are likely to discover more comprehensive refactorings and reduce the likelihood of missing commits with valid refactorings. Second, we also requested human experts to validate each identified refactoring, ensuring the reliability of the resulting dataset. The effectiveness of the employed construction strategy was also demonstrated by Tsantalis et al. (2018, 2022), who created the most comprehensive dataset with considerably reduced bias.

Although specifying expected refactoring types may increase LLMs' success rate in identifying refactoring opportunities, developers may not know exactly the expected refactoring types in advance. In this case, the developers should either try

each of the popular refactoring types one by one or add all such refactoring types as expected refactorings. As introduced in Sect. 3.4, explicitly specifying the refactoring subcategories and narrowing the search space may further improve LLMs' success rate in identifying refactoring opportunities. Similarly, developers may not know exactly the expected refactoring subcategories in advance. Therefore, we suggest that developers should try each of the proposed refactoring subcategories one by one to determine whether the given source code contains such refactoring opportunities. Moreover, the proposed refactoring subcategories in this paper may not be exhausted. Consequently, developers should customize dedicated prompts for the refactoring subcategories they are interested in, based on the prompt templates provided in this paper.

5.3 Implications

LLMs are best used as suggestive auxiliary tools rather than precise and reliable code refactoring tools. On one side, LLMs can identify many refactoring opportunities and suggest many high-quality refactoring solutions, and thus they could be used as smart assistants. On the other side, LLMs often miss refactoring opportunities, suggest nonsense solutions, and conduct refactorings incorrectly (resulting in bugs), which makes the current form of LLMs unsuitable as a reliable refactoring engine.

The study's findings demonstrate that LLMs possess the potential to autonomously identify refactoring opportunities. However, without clear and concrete guidance, LLMs may struggle to accurately pinpoint such opportunities. Therefore, we suggest that developers avoid relying on generic prompts when working with LLMs for refactoring tasks. Instead, they could systematically apply the tailored prompt templates we provide for each type of refactoring, as outlined in Sect. 3.4. By applying these prompts one by one, LLMs can be effectively guided to identify potential refactoring opportunities in a sequential and structured manner. Once a refactoring opportunity is identified, LLMs can generate corresponding refactoring suggestions, which developers can review and decide whether to implement the suggested changes. Notably, LLMs serve as only one component of the automated refactoring process. We believe that as LLM capabilities continue to advance, the proposed prompt templates and stepwise strategies will increasingly align with developers' refactoring objectives, facilitating a more efficient and targeted refactoring workflow.

6 Related work

6.1 Traditional approaches of refactorings

Refactoring is the process of improving the internal structure of source code without altering its external behavior (Fowler 1999; Mens and Tourwé 2004). Existing research on refactoring can be broadly categorized into two areas: *refactoring*

opportunity identification and *refactoring solution recommendation*. Code smells refer to potential issues or design flaws within the codebase, while refactoring serves as an effective technique to remove them (Lacerda et al. 2020). One well-known and widely studied code smell is the "long methods", which violate the Single Responsibility Principle (SRP). A common solution to this issue is the extract method refactoring, which decomposes long methods into smaller, more independent methods. Tsantalis and Chatzigeorgiou (2009, 2011) proposed an approach to identify extract method refactoring opportunities that are associated with the complete computations of a given variable declared within a method. They also utilized program slicing techniques to extract the statements that affect the state of a given object as the suggested solution. Charalampidou et al. (2017) exploited functional relevance to identify source code chunks that implement specific functionality and proposed extracting them into a new method. Similarly, god classes are another code smell that violates SRP because they typically encompass too much functionality. Bavota et al. (2010) introduced a game theory-based approach to identify extract class refactoring opportunities, i.e., the extracted two classes are more cohesive than the original class. Bavota et al. (2011) combined structural and semantic cohesion measures to identify extract class refactoring opportunities. Chen et al. (2024) developed ClassSplitter, a novel position-based approach for decomposing god classes. Their findings suggested that code entities (such as methods and fields) that are physically adjacent tend to have a higher likelihood of being grouped together.

Bavota et al. (2015) investigated the innate relationship between code quality and software refactoring. They found that highly coupled classes are more likely to involve inline method refactoring. Peruma et al. (2018) conducted an empirical study on how and why developers rename identifiers. They emphasize that identifier renaming is to improve understandability. Liu et al. (2015) proposed an approach to identify renaming refactoring opportunities by expanding conducted renaming activities. They assumed that renaming activities in commit history could uncover potential rename refactoring opportunities and assist in recommending new names. Similarly, Zhang et al. (2023) leveraged identifier features and renaming history to train a decision tree-based classifier, which was then employed to identify renaming refactoring opportunities. For those identifiers that need renaming, they extracted related code entities and their renaming history to suggest suitable new names. Liu et al. (2013) presented a monitor-based refactoring framework (named InsRefactor) to identify refactoring opportunities promptly. InsRefactor supported real-time detection of eight types of code smells, including bad names, long methods, etc.

6.2 Deep learning-based refactorings

With the advances in deep learning technologies, many researchers have investigated deep learning-based refactoring. Liu et al. (2021) were the first to apply deep learning techniques to software refactoring. Their major contribution is a novel approach to synthesizing large-scale training data for neural networks that are designed to detect code smells and suggest refactoring solutions. However, deep neural networks trained with synthetic data may only learn to identify these artificial

smells, rather than real-world code smells. To this end, Liu et al. (2023) proposed to automatically collect code smells and refactorings from real-world projects, which could be directly employed as high-quality training data. Barbez et al. (2019) were the first to exploit historical and structural information to identify god class smells with the help of deep learning techniques. Kurbatova et al. (2020) proposed a hybrid approach to identify feature envy smells that combines deep learning techniques (i.e., code2vec (Alon et al. 2019)) and an SVM-based classifier. Similarly, Cui et al. (2022) applied code2vec (and code2seq (Alon et al. 2019)) to transform methods into numeric vectors, and utilized graph embedding techniques to represent the dependencies between the method and other methods. With the resulting embeddings, they leveraged traditional machine learning techniques (like Naive Bayes) to suggest move method refactorings.

Tufano et al. (2019) conducted the first empirical study on the capability of neural machine translation (NMT) models to learn and apply code changes (e.g., refactorings) made by developers. Their evaluation results suggest that NMT models can replicate up to 36% of the changes made by developers. Liu et al. (2019) proposed an automated approach based on deep learning and information retrieval to identify methods whose names are inconsistent with the corresponding method bodies. Desai et al. (2021) proposed to automatically refactor a monolith application into multiple microservices using deep learning techniques. Liu et al. (2023) proposed a two-stage pre-trained framework based on BERT architecture (Devlin et al. 2019) to suggest appropriate names for poorly named variables. Vitale et al. (2023) proposed an approach to automatically improve code readability by fine-tuning the T5 model (Raffel et al. 2020) with readability-improved commits mined from software repositories.

6.3 LLM-based software refactoring

Given the remarkable proficiency of LLMs in various software engineering tasks, several studies (Chouchen et al. 2024; Tufano et al. 2024; AlOmar et al. 2024; Deo et al. 2024) have investigated how developers exploit LLMs for software refactoring. These studies emphasize the role of LLMs in improving software quality through refactoring and detail how developers interact with LLMs to specify and execute refactoring. Complex programs often present challenges in terms of readability and maintainability. To address these issues, Shirafuji et al. (2023) proposed an approach that uses few-shot examples to prompt LLMs to suggest simplified versions of such programs. Their evaluation results suggest that the simplified programs significantly reduce cyclomatic complexity and the number of lines in code while preserving semantic correctness. Existing Transformation by Example (TBE) techniques are employed to automate code changes. However, their effectiveness may struggle with unseen code change patterns. To this end, Dilhara et al. (2024) proposed PyCraft, which utilized LLMs to generate semantically equivalent and yet previously unseen variants of change patterns. With such variants, PyCraft could significantly increase the effectiveness of TBE techniques and expedite the software development process. Pomian et al. (2024a, 2024b) proposed an LLM-based approach, called

`EM-Assist`, to suggest safety *extract method* refactorings for decomposing long methods. As the initial step, LLMs were repeatedly requested to generate *extract method* refactoring solutions. To ensure the suggested solutions were valid, i.e., the refactored code could compile successfully, the authors utilized IDE's API to validate whether the suggestions meet the refactoring preconditions for the extract method. Subsequently, the program slicing techniques were employed to enhance the LLMs' suggestions by adjusting the code fragments. Although `EM-Assist` received favorable feedback from many industrial developers, its scope is limited to a specific type of refactoring and does not conduct a comprehensive evaluation of LLMs' refactoring capabilities. DePalma et al. (2024) explored the refactoring capabilities of ChatGPT, revealing that it can effectively refactor code and offer insights into the refactored code. However, their study lacked a quantitative evaluation of how well LLMs perform in automate software refactoring compared to human experts. To this end, we construct a high-quality dataset consisting of refactorings conducted by developers in real-world scenarios, and evaluate for the first time whether LLMs have the potential to identify refactoring opportunities accurately. Furthermore, we propose a detect-and-reapply tactic to mitigate the hallucination issues associated with LLMs, ensuring the reliability of LLM-based refactoring.

Existing studies primarily focus on using LLMs to suggest and execute specific refactorings automatically, yet there is a lack of comprehensive empirical study on the refactoring capability of LLMs compared to those of human experts. Moreover, we propose the detect-and-reapply tactic to mitigate issues with LLM-generated *hallucinations*. Notably, in this paper, the hallucinations refer to instances where LLM-driven refactorings result in syntax errors or change the intended behavior of the code. This definition differs from that of Pomian et al. (2024a, 2024b), who developed EM-Assist to decompose long methods and defined LLMs' hallucinations as refactoring suggestions that fail to meet IDE's refactoring preconditions and refactoring solutions that involve only one single-line statement or the whole method body. Our paper aims to complement these studies by conducting a rigorous empirical study on LLM-based software refactoring. Our study differs from the existing approaches in that we not only investigate the capability of LLMs in suggesting/executing refactorings, but also systematically evaluate the capability of LLMs in identifying refactoring opportunities within the given source code.

7 Conclusions and future work

In this paper, we conduct an empirical study to investigate the potential of LLM-based software refactoring. Our findings suggest that with generic prompts, the performance of LLMs in identifying refactoring opportunities is generally low, highlighting an ongoing challenge in this area. However, with proper prompts, LLMs have the potential to identify up to 86.7% of the refactoring opportunities with high accuracy. LLMs also have the potential to suggest high-quality refactoring solutions for the identified refactoring opportunities, and such solutions are comparable to those constructed manually by human experts. However, our evaluation results also suggest that LLMs may introduce some dangerous changes that either reduce the

performance of the source code or introduce function and/or syntax errors. To this end, in this paper, we propose a detect-and-reapply tactic to void buggy changes. Our evaluation results suggest that it successfully avoided all bugs initially introduced by the evaluated LLMs.

Our empirical study demonstrates the potential of LLM-based refactoring, which may inspire further research in this line. It could be interesting in the future to investigate how to assess the refactoring opportunities and refactoring solutions suggested by LLMs, and thus we can pick up only beneficent refactoring opportunities and high-quality refactoring solutions. It is also potentially fruitful to investigate how to improve LLM-based refactoring by pre-processing and post-processing. For example, according to the few-shot learning techniques, it could be helpful to retrieve examples from a corpus according to the to-be-refactored source code and append them to the prompt. Our evaluation results may also encourage the integration of LLM-based refactoring into code review. The LLMs' suggestions could be fed to developers, and they would manually validate such suggestions and conduct refactorings, which helps avoid unsafe refactorings.

Acknowledgements This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037), China National Postdoctoral Program for Innovative Talents (BX20240008) and CCF-Huawei Populus Grove Fund (CCF-HuaweiSE202411).

Data Availability The replication package, including the tools and the data, is publicly available (Liu 2024b).

Declarations

Conflict of interest The authors declare no conflict of interest.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: GPT-4 technical report (2023). https://arxiv.org/abs/2303.08774

Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: generating sequences from structured representations of code. In: Proceedings of the 7th International Conference on Learning Representations (ICLR'19). OpenReview, New Orleans, LA, USA (2019)

Alizadeh, V., Kessentini, M.: Reducing interactive refactoring effort via clustering-based multi-objective search. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18), pp. 464–474. ACM, Montpellier (2018). https://doi.org/10.1145/3238147.3238217

Alizadeh, V., Kessentini, M., Mkaouer, M.W., Cinnéide, M.Ó., Ouni, A., Cai, Y.: An interactive and dynamic search-based approach to software refactoring recommendations. IEEE Trans. Softw. Eng. **46**(9), 932–961 (2020). https://doi.org/10.1109/TSE.2018.2872711

Akın, F.K.: Awesome ChatGPT prompts (2024). https://github.com/f/awesome-chatgpt-prompts

AlOmar, E.A., Mkaouer, M.W., Ouni, A.: Behind the intent of extract method refactoring: a systematic literature review. IEEE Trans. Softw. Eng. (2024). https://doi.org/10.1109/TSE.2023.3345800

Alizadeh, V., Ouali, M.A., Kessentini, M., Chater, M.: RefBot: intelligent software refactoring bot. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19), pp. 823–834. IEEE, San Diego, CA, USA (2019). https://doi.org/10.1109/ASE.2019.00081

AlOmar, E.A., Venkatakrishnan, A., Mkaouer, M.W., Newman, C.D., Ouni, A.: How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In: Proceedings of

the 21st International Conference on Mining Software Repositories (MSR'24), pp. 202–206. IEEE (2024). https://doi.org/10.1145/3643991.3645081

Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. In: Proceedings of the ACM on Programming Languages 3(POPL), pp. 1–29 (2019). https://doi.org/10.1145/3291636

Baqais, A.A.B., Alshayeb, M.: Automatic software refactoring: a systematic literature review. Softw. Qual. J. **28**(2), 459–502 (2020). https://doi.org/10.1007/s11219-019-09477-y

Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W., et al.: A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity (2023). arxiv: 2302.04023

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F.: An experimental investigation on the innate relationship between quality and refactoring. J. Syst. Softw. **107**, 1–14 (2015). https://doi.org/10.1016/j.jss.2015.05.024

Bavota, G., De Lucia, A., Oliveto, R.: Identifying extract class refactoring opportunities using structural and semantic cohesion measures. J. Syst. Softw. **84**(3), 397–414 (2011). https://doi.org/10.1016/j.jss.2010.11.918

Barbez, A., Khomh, F., Guéhéneuc, Y.-G.: Deep learning anti-patterns from code metrics history. In: Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME'19), pp. 114–124. IEEE, Cleveland, OH, USA (2019). https://doi.org/10.1109/ICSME.2019.00021

Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., Guéhéneuc, Y.-G.: Playing with refactoring: identifying extract class opportunities through game theory. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM'10), pp. 1–5. IEEE (2010). https://doi.org/10.1109/ICSM.2010.5609739

Charalampidou, S., Ampatzoglou, A., Chatzigeorgiou, A., Gkortzis, A., Avgeriou, P.: Identifying extract method refactoring opportunities based on functional relevance. IEEE Trans. Softw. Eng. **43**(10), 954–974 (2017). https://doi.org/10.1109/TSE.2016.2645572

Chouchen, M., Bessghaier, N., Begoug, M., Ouni, A., AlOmar, E.A., Mkaouer, M.W.: How do so ware developers use ChatGPT? An exploratory study on github pull requests. In: Proceedings of the 21st International Conference on Mining Software Repositories (MSR'24), pp. 212–216. IEEE (2024). https://doi.org/10.1145/3643991.3645084

Chang, S., Fosler-Lussier, E.: How to prompt LLMs for text-to-SQL: a study in zero-shot, single-domain, and cross-domain settings (2023). arxiv: 2305.11853

Chen, T., Jiang, Y., Fan, F., Liu, B., Liu, H.: A position-aware approach to decomposing god classes. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE'24), pp. 129–140 (2024). IEEE. https://doi.org/10.1145/3691620.3694992

Cui, D., Wang, S., Luo, Y., Li, X., Dai, J., Wang, L., Li, Q.: RMove: recommending move method refactoring opportunities using structural and semantic representations of code. In: Proceedings of the 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME'22), pp. 281–292. IEEE, Limassol, Cyprus (2022). https://doi.org/10.1109/ICSME55016.2022.00033

Dair.AI: Prompt Engineering Guide (2024). https://github.com/dair-ai/Prompt-Engineering-Guide/blob/main/guides/prompts-intro.md

Dawes, J.: Do data characteristics change according to the number of scale points used? An experiment using 5-point, 7-point and 10-point scales. Int. J. Mark. Res. **50**(1), 61–104 (2008). https://doi.org/10.1177/147078530805000106

Dilhara, M., Bellur, A., Bryksin, T., Dig, D.: Unprecedented code change automation: the fusion of LLMS and transformation by example. In: Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE'24), pp. 631–653. ACM, Porto de Galinhas, Brazil (2024). https://doi.org/10.1145/3643755

Desai, U., Bandyopadhyay, S., Tamilselvam, S.: Graph neural network to dilute outliers for refactoring monolith application. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21), vol. 35, pp. 72–80 (2021). https://doi.org/10.1609/aaai.v35i1.16079

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'19), pp. 4171–4186. ACL, Minneapolis, MN, USA (2019). https://doi.org/10.18653/V1/N19-1423

Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06), pp. 404–428. Springer, Nantes, France (2006). https://doi.org/10.1007/11785477_24

Deo, S., Hinge, D., Chavan, O.S., Wang, Y.O., Mkaouer, M.W.: Analyzing developer-ChatGPT conversations for software refactoring: an exploratory study. In: Proceedings of the 21st International Conference on Mining Software Repositories (MSR'24), pp. 207–211 (2024). IEEE. https://doi.org/10.1145/3643991.3645082

Dice, L.R.: Measures of the amount of ecologic association between species. Ecology **26**(3), 297–302 (1945). https://doi.org/10.2307/1932409

DePalma, K., Miminoshvili, I., Henselder, C., Moss, K., AlOmar, E.A.: Exploring ChatGPT's code refactoring capabilities: an empirical study. Expert Syst. Appl. **249**, 1–26 (2024). https://doi.org/10.1016/j.eswa.2024.123602

Farmmamba: Hadoop HDFS. https://issues.apache.org/jira/browse/HDFS-17322 (2024)

Fontana, F.A., Caracciolo, A., Zanoni, M.: DPB: A benchmark for design pattern detection tools. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12), Szeged, Hungary, pp. 235–244. IEEE (2012). https://doi.org/10.1109/CSMR.2012.32

Fulop, L.J., Ferenc, R., Gyimóthy, T.: Towards a benchmark for evaluating design pattern miner tools. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), pp. 143–152. IEEE, Athens, Greece (2008). https://doi.org/10.1109/CSMR.2008.4493309

Foster, S.R., Griswold, W.G., Lerner, S.: WitchDoctor: IDE support for real-time auto-completion of refactorings. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), pp. 222–232. IEEE, Zurich, Switzerland (2012). https://doi.org/10.1109/ICSE.2012.6227191

Fleiss, J.L.: Measuring nominal scale agreement among many raters. Psychol. Bull. **76**(5), 378–382 (1971). https://doi.org/10.1037/h0031619

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14), pp. 313–324. ACM, Vasteras, Sweden (2014). https://doi.org/10.1145/2642937.2642982

Feitelson, D.G., Mizrahi, A., Noy, N., Shabat, A.B., Eliyahu, O., Sheffer, R.: How developers choose names. IEEE Trans. Softw. Eng. **48**(1), 37–52 (2022). https://doi.org/10.1109/TSE.2020.2976920

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

Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A.: JDeodorant: identification and removal of feature envy bad smells. In: Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM'07), pp. 519–520. IEEE, Paris, France (2007). https://doi.org/10.1109/ICSM.2007.4362679

Fluri, B., Wursch, M., Pinzger, M., Gall, H.: Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. **33**(11), 725–743 (2007). https://doi.org/10.1109/TSE.2007.70731

Grund, F., Chowdhury, S.A., Bradley, N.C., Hall, B., Holmes, R.: CodeShovel: constructing method-level source code histories. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21), pp. 1510–1522. IEEE, Madrid, Spain (2021). https://doi.org/10.1109/ICSE43902.2021.00135

Guo, Q., Cao, J., Xie, X., Liu, S., Li, X., Chen, B., Peng, X.: Exploring the potential of ChatGPT in automated code refinement: an empirical study. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE'24), pp. 1–13. ACM, Lisbon, Portugal (2024). https://doi.org/10.1145/3597503.3623306

Ge, X., DuBose, Q.L., Murphy-Hill, E.: Reconciling manual and automatic refactoring. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), pp. 211–221. IEEE, Zurich, Switzerland (2012). https://doi.org/10.1109/ICSE.2012.6227192

Grissom, R.J., Kim, J.J.: Effect Sizes for Research: A Broad Practical Approach. Lawrence Erlbaum Associates, Hillsdale, NJ (2005)

Google: Gemini Model (2024). https://cloud.google.com/vertex-ai/generative-ai/docs/learn/models

JetBrains: IntelliJ IDEA Refactoring Meun (2024). https://www.jetbrains.com/help/idea/refactoring-source-code.html

Kniesel, G., Binun, A.: Standing on the shoulders of giants—a data fusion approach to design pattern detection. In: Proceedings of the IEEE 17th IEEE International Conference on Program

Comprehension (ICPC'09), pp. 208–217. IEEE, Vancouver, BC, Canada (2009). https://doi.org/10.1109/ICPC.2009.5090044

Kim, M., Gee, M., Loh, A., Rachatasumrit, N.: Ref-Finder: A refactoring reconstruction tool based on logic query templates. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10), pp. 371–372. ACM, Santa Fe, NM, USA (2010). https://doi.org/10.1145/1882291.1882353

Kurbatova, Z., Veselov, I., Golubev, Y., Bryksin, T.: Recommendation of move method refactoring using path-based representation of code. In: Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering Workshops (IWoR'20), pp. 315–322. ACM, Seoul, Republic of Korea (2020). https://doi.org/10.1145/3387940.3392191

Liu, H., Guo, X., Shao, W.: Monitor-based instant software refactoring. IEEE Trans. Softw. Eng. **39**(8), 1112–1126 (2013). https://doi.org/10.1109/TSE.2013.4

Liu, Y., Han, T., Ma, S., Zhang, J., Yang, Y., Tian, J., He, H., Li, A., He, M., Liu, Z., et al.: Summary of ChatGPT-related research and perspective towards the future of large language models. Meta-Radiology **1**(2), 1–14 (2023). https://doi.org/10.1016/j.metrad.2023.100017

Liu, B.: ReExtractor (2024). https://github.com/lyoubo/ReExtractor

Liu, B.: Replication Package (2024). https://github.com/bitselab/LLM4Refactoring

Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., Zhang, L.: Deep learning based code smell detection. IEEE Trans. Softw. Eng. **47**(9), 1811–1837 (2021). https://doi.org/10.1109/TSE.2019.2936376

Liu, K., Kim, D., Bissyandé, T.F., Kim, T., Kim, K., Koyuncu, A., Kim, S., Le Traon, Y.: Learning to spot and refactor inconsistent method names. In: Proceedings of the 41st International Conference on Software Engineering (ICSE'19), pp. 1–12. IEEE, Montreal, QC, Canada (2019). https://doi.org/10.1109/ICSE.2019.00019

Liu, B., Liu, H., Li, G., Niu, N., Xu, Z., Wang, Y., Xia, Y., Zhang, Y., Jiang, Y.: Deep learning based feature envy detection boosted by real-world examples. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23), pp. 908–920. ACM, San Francisco, CA, USA (2023). https://doi.org/10.1145/3611643.3616353

Liu, H., Liu, Q., Liu, Y., Wang, Z.: Identifying renaming opportunities by expanding conducted rename refactorings. IEEE Trans. Softw. Eng. **41**(9), 887–900 (2015). https://doi.org/10.1109/TSE.2015.2427831

Liu, B., Liu, H., Niu, N., Zhang, Y., Li, G., Jiang, Y.: Automated software entity matching between successive versions. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23), pp. 1615–1627. IEEE, Luxembourg, Luxembourg (2023). https://doi.org/10.1109/ASE56229.2023.00132

Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.G.: Code smells and refactoring: a tertiary systematic review of challenges and observations. J. Syst. Softw. **167**, 110610 (2020). https://doi.org/10.1016/j.jss.2020.110610

Liu, H., Wang, Y., Wei, Z., Xu, Y., Wang, J., Li, H., Ji, R.: RefBERT: a two-stage pre-trained framework for automatic rename refactoring. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23), pp. 740–752. ACM, Seattle, WA, USA (2023). https://doi.org/10.1145/3597926.3598092

Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2012). https://doi.org/10.1109/TSE.2011.41

Mkaouer, M.W., Kessentini, M., Cinnéide, M.Ó., Hayashi, S., Deb, K.: A robust multi-objective approach to balance severity and importance of refactoring opportunities. Empir. Softw. Eng. **22**, 894–927 (2017). https://doi.org/10.1007/s10664-016-9426-8

Minna, F., Massacci, F., Tuma, K.: Analyzing and Mitigating (with LLMs) the Security Misconfigurations of Helm Charts from Artifact Hub (2024). https://arxiv.org/abs/2403.09537

Mu, F., Shi, L., Wang, S., Yu, Z., Zhang, B., Wang, C., Liu, S., Wang, Q.: ClarifyGPT: empowering LLM-based code generation with intention clarification (2023). https://arxiv.org/abs/2310.10996

Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004). https://doi.org/10.1109/TSE.2004.1265817

Negara, S., Chen, N., Vakilian, M., Johnson, R.E., Dig, D.: A comparative study of manual and automated refactorings. In: Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13), pp. 552–576. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39038-8_23

OpenAI: ChatGPT (2024). https://openai.com/index/chatgpt

OpenAI: GPT-4 Model (2024). https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Bryksin, T., Dig, D.: Together We Go Further: LLMs and IDE Static Analysis for Extract Method Refactoring (2024). arxiv: 2401.15298

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Sokolov, A., Bryksin, T., Dig, D.: EM-Assist: safe automated extract method refactoring with LLMs. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE'24'), pp. 582–586. ACM (2024). https://doi.org/10.1145/3663529.3663803

PMD: PMD (2024). https://github.com/pmd/pmd

Peruma, A., Mkaouer, M.W., Decker, M.J., Newman, C.D.: An empirical investigation of how and why developers rename identifiers. In: Proceedings of the 2nd International Workshop on Refactoring (IWoR'18'), pp. 26–33. ACM (2018). https://doi.org/10.1145/3242163.3242169

Prete, K., Rachatasumrit, N., Sudan, N., Kim, M.: Template-based reconstruction of complex refactorings. In: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10), pp. 1–10. IEEE, Timisoara, Romania (2010). https://doi.org/10.1109/ICSM.2010.5609577

Peruma, A., Simmons, S., AlOmar, E.A., Newman, C.D., Mkaouer, M.W., Ouni, A.: How do I refactor this? An empirical study on refactoring trends and topics in Stack Overflow. Empir. Softw. Eng. **27**(11), 1–43 (2022). https://doi.org/10.1007/s10664-021-10045-x

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. **21**(1), 5485–5551 (2020). https://doi.org/10.18653/V1/N19-1423

Silva, D., Silva, J.P., Santos, G., Terra, R., Valente, M.T.: RefDiff 2.0: A multi-language refactoring detection tool. IEEE Trans. Softw. Eng. **47**(12), 2786–2802 (2020). https://doi.org/10.1109/TSE.2020.2968072

Shirafuji, A., Oda, Y., Suzuki, J., Morishita, M., Watanobe, Y.: Refactoring programs using large language models with few-shot examples. In: Proceedings of the 30th Asia-Pacific Software Engineering Conference (APSEC'23), pp. 151–160. IEEE (2023). https://doi.org/10.1109/APSEC60848.2023.00025

Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? Confessions of GitHub contributors. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16), pp. 858–870. ACM, Seattle WA USA (2016). https://doi.org/10.1145/2950290.2950305

Silva, D., Valente, M.T.: RefDiff: detecting refactorings in version histories. In: Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17), pp. 269–279. IEEE (2017). https://doi.org/10.1109/MSR.2017.14

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A.M., Hauth, A., et al.: Gemini: a family of highly capable multimodal models (2023). arXiv. 2312.11805

Tate, R.F.: Correlation between a discrete and a continuous variable, point-biserial correlation. Ann. Math. Stat. **25**(3), 603–607 (1954). https://doi.org/10.1214/aoms/1177728730

Tsantalis, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Trans. Softw. Eng. **35**(3), 347–367 (2009). https://doi.org/10.1109/TSE.2009.1

Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. J. Syst. Softw. **84**(10), 1757–1782 (2011). https://doi.org/10.1016/j.jss.2011.05.016

Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: JDeodorant: Identification and removal of type-checking bad smells. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), pp. 329–331. IEEE, Athens, Greece (2008). https://doi.org/10.1109/CSMR.2008.4493342

Tsantalis, N., Ketkar, A., Dig, D.: RefactoringMiner 2.0. IEEE Trans. Softw. Eng. **48**(3), 930–950 (2022). https://doi.org/10.1109/TSE.2020.3007722

Tourwé, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03), pp. 91–100. IEEE, Benevento, Italy (2003). https://doi.org/10.1109/CSMR.2003.1192416

Tsantalis, N., Mansouri, M., Eshkevari, L.M., Mazinanian, D., Dig, D.: Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th International Conference on Software Engineering (ICSE'18), pp. 483–494. ACM, Gothenburg, Sweden (2018). https://doi.org/10.1145/3180155.3180206

Tufano, R., Mastropaolo, A., Pepe, F., Dabić, O., Di Penta, M., Bavota, G.: Unveiling ChatGPT's usage in open source projects: A mining-based study. In: Proceedings of the 21st International Conference

on Mining Software Repositories (MSR '24), pp. 571–583. IEEE (2024). https://doi.org/10.1145/3643991.3644918

Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., Poshyvanyk, D.: On learning meaningful code changes via neural machine translation. In: Proceedings of the 41st International Conference on Software Engineering (ICSE'19), pp. 25–36. IEEE, Montreal, QC, Canada (2019). https://doi.org/10.1109/ICSE.2019.00021

Vitale, A., Piantadosi, V., Scalabrino, S., Oliveto, R.: Using deep learning to automatically improve code readability. In: Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23), pp. 573–584. IEEE, Luxembourg, Luxembourg (2023). https://doi.org/10.1109/ASE56229.2023.00112

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C.: A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT (2023). https://arxiv.org/abs/2302.11382

White, J., Hays, S., Fu, Q., Spencer-Smith, J., Schmidt, D.C.: ChatGPT prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, pp. 71–108 (2024). https://doi.org/10.1007/978-3-031-55642-5_4

Wilcoxon, F.: Individual comparisons by ranking methods. Int. Biomet. Soc. **1**(6), 80–83 (1945). https://doi.org/10.2307/3001968

Wu, Y., Li, Z., Zhang, J.M., Papadakis, M., Harman, M., Liu, Y.: Large language models in fault localisation (2023). https://arxiv.org/abs/2308.15276

Xing, Z., Stroulia, E.: The JDEvAn tool suite in support of object-oriented evolutionary development. In: Companion of the 30th International Conference on Software Engineering (ICSE Companion'08), pp. 951–952. ACM, Leipzig, Germany (2008). https://doi.org/10.1145/1370175.1370203

Xia, C.S., Zhang, L.: Keep the Conversation Going: Fixing 162 out of 337 bugs for \$0.42 each using ChatGPT (2023). arxiv. 2304.00385

Yaron: TestMe (2024). https://github.com/wrdv/testme-idea

Yamashita, A., Moonen, L.: Do developers care about code smells? An exploratory survey. In: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13), pp. 242–251. IEEE (2013). https://doi.org/10.1109/WCRE.2013.6671299

Zhang, J., Luo, J., Liang, J., Gong, L., Huang, Z.: An accurate identifier renaming prediction and suggestion approach. ACM Trans. Softw. Eng. Methodol. **32**(6), 1–51 (2023). https://doi.org/10.1145/3603109

Authors and Affiliations

Bo Liu[1] · Yanjie Jiang[1,2] · Yuxia Zhang[1] · Nan Niu[3] · Guangjie Li[4] · Hui Liu[1]

✉ Yanjie Jiang
yanjiejiang@pku.edu.cn

✉ Hui Liu
liuhui08@bit.edu.cn

Bo Liu
liubo@bit.edu.cn

Yuxia Zhang
yuxiazh@bit.edu.cn

Nan Niu
nan.niu@uc.edu

Guangjie Li
liguangjie_er@126.com

[1] School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China

[2] Key Laboratory of High Confidence Software Technologies, Ministry of Education, School of Computer Science, Peking University, Beijing 100871, China

[3] Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati 45221, USA

[4] National Innovation Institute of Defense Technology, Beijing 100071, China