Software Refactoring Research with Large Language Models: A Systematic Literature Review

Sofia Martinez[a,1], Luo Xu[a,1], Mariam Elnaggar[a], Eman Abdullah AlOmar[a,*]

[a]*Stevens Institute of Technology, Hoboken, NJ, USA*

Abstract

Background: Code refactoring is the improvement of code internally without changing the external functionalities of the program. Due to its exhaustive nature, developers often avoid manually refactoring code. Researchers have since looked into utilizing Large Language Models (LLMs) to automate the task of refactoring.

Aim and Method: Despite the promising results, there is a lack of clear understanding of LLMs' effectiveness in automated refactoring. In order to address this issue, we conducted a Systematic Literature Review (SLR) of 50 primary studies. We categorized the studies into different refactoring methods studied, prompt engineering and techniques conducted, LLM tools used, languages used, and datasets used. We touched upon the benchmarks each studies had used, how accurate LLM-generated refactorings are, and the challenges that this field faces currently.

Result: From our literature review we found that: (i) there are various tools that different studies use to enhance and study LLM-driving refactoring, with tools that were used to detect code smells, generate code bases, and compare refactoring outcomes. (ii) Various datasets were collected from multiple open-source projects in multiple programming languages for analysis. These platforms included GitHub, Apache, and F-Droid, with the most popular language collected and analyzed being Java. (iii) One-Shot, Few-Shot, Context-Specific, and Chain-of-Thought prompting methods have been shown to be the most effective depending on the language used. In some instances, being capable of reducing code smell by up to 89%. (iv) The definition of "Accuracy" varies significantly across the literature surveyed, as this depends on the context of the study, thus calling for a need to have a standardized measurement for Accuracy. (v) The most often mentioned code smells were Large Class and Long Method, with a lot of studies also not specifying, while the most often applied refactoring type is Extract Method, showing promising results in using LLM to perform this refactoring type. (vi) When working with LLMs, they often generate erro-

*Corresponding author

Email addresses: smartine1@stevens.edu (Sofia Martinez), lxu41@stevens.edu (Luo Xu), melnagga@stevens.edu (Mariam Elnaggar), ealomar@stevens.edu (Eman Abdullah AlOmar)

[1]Authors contributed equally

neous code, struggle with more complex refactoring, and often misunderstand the developers' requests and miss refactoring requests.

Conclusion: Our study serves to be a collection of knowledge on the topic of LLM refactoring found by various other studies, and to highlight any issues that are often missed by the researchers. We hope to empower and guide the future development of LLM-driven refactoring with our findings.

Keywords: refactoring, quality, literature review.

---

1. Introduction

As Large Language Models (LLMs) become more complex and capable, developers now seek to incorporate these tools into their developmental processes. One of the most important tasks in development that is often overlooked is refactoring. Refactoring refers to the activity of taking preexisting code and changing its internal structure to improve it without changing its functionalities [1, 2]. Code refactoring is imperative to the readability, maintainability, and efficiency of the program, especially for legacy code bases that have grown over time. This code held a large amount of technical debt that many human developers would avoid fixing, and rather, focus on code development over maintainability or efficiency [3]. Due to this habit, when developers do have time to go back to fix the existing code base, it often ends up taking them a substantial amount of time.

Therefore, previous studies have started exploring how to automate code refactoring to build better code and to cut down on manual labor [4]. Many studied the effects of external frameworks and tools used alongside LLMs [5–12], the kinds of refactoring done by LLMs [13–18], and the different ways to prompt LLMs [13–17]. However, there is a lack of Systematic Literature Review (SLR) on evaluating the overarching impacts of LLMs on automated code refactoring. Oftentimes, literature exists to address only one part of the problem in LLM-developer interactions (such as analyzing Few-Shot prompting [15] or analyzing effect of LLMs with Java refactoring [9]).

The specifications of the studies also needed to be addressed; this study isolates the evaluation of refactoring by LLMs as a separate case compared to using LLMs in other ways as code refactoring holds special usages for models. In using LLMs for refactoring, one must consider the added layer of how the code will exist within a predefined environment while also replicating what already previously existed. In other words, how can developers interact with LLMs to get the perfect response, that is the perfect blend of style, utility, and maintainability? This approach in refactoring addresses several aspects of LLMs' capabilities that LLMs have shown to be capable of while also showing errors in; thus we must address what are these errors and how to best maximize the usage of LLMs in this case.

This SLR can then help address questions across the board in different programming languages and the best techniques for interacting with LLMs as a

developer for refactoring instead of being limited to one topic, tool, method or language. To address this issue, we reviewed the most relevant studies done with LLMs-driven refactoring. We extracted key information from 50 studies and categorized the papers, identifying the most often used techniques by researchers in their evaluations–such as prompting strategies, evaluation methods, types of refactorings performed–and identified the major challenges faced when automating refactoring using LLMs. We found that prompt structure, context of the project, and providing examples can significantly improve the quality of LLM-generated refactorings, mitigating critical concerns such as hallucinations, context loss, and scalability issues. We hope that this SLR can help bridge the gaps between studies and further the study on the topic of LLM refactoring.

When defining our research questions, we framed them around the quantitative and qualitative aspects of the papers as follows:

- **RQ_1: What are the tools used in LLM-driven refactoring?** We posed this question to categorize and study the effects of the existing approaches and tools for LLM-driven refactoring. When implementing different tools, frameworks, and approaches alongside LLMs, we see a considerably large amount of changes in output.

- **RQ_2: What datasets and benchmarks have been used to evaluate LLMs for refactoring tasks?** We posed this question to examine the various datasets and benchmarks used to evaluate LLMs for refactoring and further understand the design of the experiments in each study.

- **RQ_3: What prompt techniques have been used to perform refactoring tasks?** We posed this question to examine what prompting methods researchers have been exploring. By employing the best prompting strategies, we can help reduce hallucinations and maximize refactoring efficiency.

- **RQ_4: How accurate are LLM-generated refactoring suggestions?** This RQ investigates the Accuracy of refactoring suggestions generated by LLMs. To this end, we perform quantitative, qualitative, comparative, and correctness data analysis of the LLM-based tools and approaches for each study.

- **RQ_5: What refactoring characteristics are addressed in LLM-driven refactoring?** This RQ investigates the main attributes of refactoring in LLM-assisted refactoring through the lens of 4 identified data collection: qualitative, quantitative, comparative, and correctness.

- **RQ_6: What are the key challenges in automating refactoring using LLMs?** We pose this question to identify and understand the difficulties that arise with LLM-based refactoring and analyze possible methods to address them.

The contributions of this paper are summarized as follows:

- A comprehensive analysis of the existing literature on LLM-based refactoring across tool supports, datasets used, prompt engineering techniques, and refactoring types.
- A breakdown of LLM-driven refactoring approaches and an outline of strengths and applications of the major refactoring methods.
- Identifying open issues in existing research and recommendations for future research directions, including concerns, prompt engineering strategies, tools, and refactoring methods implementations.

The remainder of the paper is structured as follows: Section 2, a review of existing studies related to LLM-driven refactoring. Section 3, an outline of our study setup in terms of search strategy and study selections. Section 4, the findings of our research and analysis. Section 5, concerns and call-to-action for possible future research. Section 6, the threats to the validity of our work, and concluded with Section 7.

2. Related Work

Baqais and Alshayeb [4] presented a systematic literature review on automated software refactoring. The authors applied multiple filtering steps to identify 41 relevant studies. Their analysis highlights growing interest in automated and search-based code refactoring, but model refactoring remains underexplored. Singh and Kaur [19] conducted a systematic literature survey on 238 publications related to refactoring with respect to code smells. The authors generally covered refactoring, but were more focused on code smells and antipatterns. The study revealed various data sets and tools that are associated with performing software refactoring. AlDallal and Abdin [20] reviewed 76 selected primary studies on the impact of refactoring on different software quality attributes. The results show that different refactoring methods can have opposite impacts on different quality attributes, indicating that refactoring does not necessarily always increase all attributes of the quality of the software. The authors also used a vote-counting approach to determine the level of consistency among the results of different studies on the effect of refactoring on software quality, which sheds light on how specific refactoring methods affect some internal quality attributes. However, due to insufficient findings, the study did not identify the impacts of refactoring methods on external and other internal attributes.

Zhang *et al.* [21] investigated the effectiveness of using code bad smells to direct code refactoring. To explore this, the authors performed a systematic review of 319 articles published between 2000 and 2009, of which 39 extremely relevant studies were analyzed. The study revealed that Duplicated Code received the most research attention, while other types of code bad smells, such as Message Chains, were not studied as much, indicating insufficient knowledge of certain code bad smells. Additionally, most studies focus on finding code bad smells instead of evaluating their impacts, which suggests that there is

little evidence to support the use of Code Bad Smells to help guide software refactoring. AlOmar *et al.* [22] presented a systematic literature review on Extract Method refactoring, a popular technique for improving code quality. To classify and evaluate Extract Method research, the authors compiled 1,367 papers and filtered them down to 83 primary studies. Their findings indicate that several methods have been proposed, with 38.6% of publications focusing on code clones, some of the tools requiring developer involvement, and since existing benchmarks are not standardized, making comparisons between tools is difficult.

More recently, Jiang *et al.* [23] surveyed the role of Large Language Models in code generation, giving a systematic overview of how LLMs can be applied to software development tasks such as transformation, synthesis, and completion. While the focus of the paper was not explicitly on refactoring, many of the identified challenges, such as ensuring semantic preservation, handling complex code structures, and evaluating code quality, are similar to the issues faced in automated refactoring. This connection suggests that LLMs face common underlying challenges across a wide range of applications within software engineering. He *et al.* [24] presented a systematic literature review on the usage of LLM agents in software development with 71 published papers and provided a breakdown of how LLM-powered Multi-Level Agents (LMAs) worked and what are some existing usages for LMAs. The authors found that there are different ways of setting up LMAs, such as having agents working in agile or waterfall models, having agents being capable to cross-examine different answers and check on its own reasoning, making LMAs less prone to mistakes and hallucinations. The authors has also found that LMAs are useful across different software engineering fields such as requirements engineering, code generation, software QA, and maintenance; but for future studies, it should be examined how can agents be best trained for different specialized roles and how show tasks be delegated between LMAs and human developers. Hou *et al.* [25] reviewed 395 papers from 2017 - 2024, the paper reported seeing over 70 different LLMs being used for software engineering tasks, and these LLMs can be separated into 3 different kinds of architectures: Decoder-only, Encoder-only, and Decoder-Encoder. Out of these 3, Decoder-only models are seen to be the most flexible and are used for both code generation and completion. In the datasets examined, the author reported seeing open-source datasets being the most prevalent, at 62.83% of the datasets collected, calling into question the need to see more industry datasets. The authors have stated what tasks in software engineering LLMs are most often tasked to complete, which are categorized into: Requirements engineering, Software design, Software development, Software quality assurance, Software maintenance, and Software management, with Software development having the most references. The authors have found there to be a lack of consistency and standard across the industry, making it hard to come to conclusive evaluations across studies.

Zapkus and Slotkienė [26] conducted a systematic literature review on the usage of LLMs on creating unit tests, the authors reported that the LLMs Codex, GPT-3, StarCoder, and GPT-4 were used to generate unit tests. The

authors reported that the topic of LLM code generation is on the rise, increasing by 56.82% per year. Wang and Chen [27] conducted a systematic literature review with 20 existing papers and found that the research that goes into LLM-generated code fails to keep up with the fast-moving pace of the applications, and that there is a lack of standardization of quality assessment along with lack of assessment beyond security and safety, calling these two qualities the baseline. The authors also pointed out that there is a lack of consideration for the human developer when conducting the evaluation. Husein *et al.* [28] conducted a systematic literature review with 23 papers and found that LLMs can perform code generation/completion on different levels, where token level is the most widely studied. They have found that with bigger datasets and transformer-based models, Accuracy is increased. They also report that dynamically typed languages, such as python, are more prone to error than static languages such as Java when generated by LLMs.

3. Study Design

3.1. Survey Planning

3.1.1. Identifying the Need for a Systematic Literature Review

Since the use of LLMs for refactoring is still a developing field, a comprehensive review of the application of LLMs for refactoring is needed. Although previous studies have explored various aspects of refactoring, including quality impact and code smell detection, few have specifically focused on the use of LLMs for refactoring. Therefore, the primary objectives of this study are to:

- Gather existing research on the application of LLMs for refactoring.
- Analyze research findings and conclusions to identify benefits and limitations of LLM-based refactoring.
- Highlight gaps and challenges to guide future research in this evolving field.

3.1.2. Specifying the Research Questions

During the process of conducting an SLR, it is of paramount importance to pinpoint pertinent research questions that have the potential to provide clear answers. We identified six such research questions:

- RQ_1. What are the tools used in LLM-driven refactoring?
- RQ_2. What datasets and benchmarks have been used to evaluate LLMs for refactoring tasks?
- RQ_3. What prompt techniques have been used to perform refactoring tasks?
- RQ_4. How accurate are LLM-generated refactoring suggestions?

- RQ_5. What refactoring characteristics are addressed in LLM-driven refactoring?
- RQ_6. What are the key challenges in automating refactoring using LLMs?

3.2. Primary Studies Selection

3.2.1. Search Strategy

To gather relevant studies, we searched eight digital libraries (i.e., ScienceDirect, Scopus, Springer Link, Web of Science, ACM Digital Library, IEEE Xplore, Wiley, and Google Scholar) that store published computing research papers. Given the rapid development of LLMs, we limited our search to studies published between Nov 2022 and Sept 2025. To ensure that we included research that combines both refactoring and LLMs, we constructed a search string that combines terms related to both fields and applied it to the abstract and keyword fields. TextBox 1 shows our search string in these search engines. To get a high-level picture of the covered topics, we generated a word cloud[2] of paper abstracts, as depicted in Figure 1.

> “refactor*” OR “code refinement” OR “code restructure” OR “transforming code” OR “code modification” combined with “LLM*” OR “language model” OR “language models” OR “GenAI” OR “Gen AI” OR “Generative AI” OR “GenerativeAI” OR “ChatGPT” OR “GPT” OR “pre-trained” OR “BERT” OR “Copilot”

TextBox 1: Search string.

However, some changes were made due to variations in search functionality across databases:

- In Google Scholar, filtering by abstract and keyword was not possible, so we conducted a title-based search instead. To further restrict the results and improve the relevance of retrieved studies, we limited our search terms to "refactor*", "LLM*","language models", and "language agents". Since Google Scholar does not support the wildcard (*) operator, we explicitly included each variation of the relevant terms.
- In ScienceDirect, filtering by abstract and keyword also includes the title field. Because this search mode does not support wildcard (*) operators and permits only up to eight Boolean operators, we explicitly included each variation of the relevant terms and divided the full search string into three separate queries to capture all relevant combinations of refactoring and LLM related terms.

[2]The figure is generated using Manus.

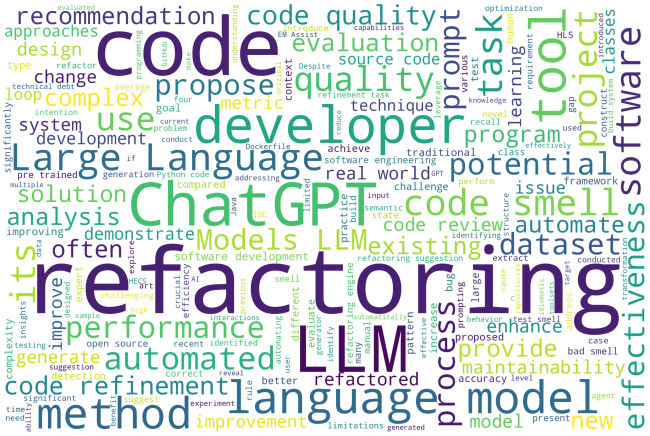

Figure 1: Word cloud of paper abstracts of primary studies.

- In Springer, searching the abstract was not possible, so we performed only a keyword search.
- In Wiley, we performed a full-text search due to the limited number of results when searching only the abstract and keywords fields.

3.2.2. Study Selection

We applied a set of inclusion and exclusion criteria, so we could choose papers that would be useful for our search. We included papers that were peer-reviewed and published in journals or conference proceedings within the field of computing. The studies also need to explicitly discuss the use of LLMs in software refactoring, not just refactoring or LLMs, to maintain our focus. We excluded non-peer-reviewed papers, such as preprints [8, 29–31], papers written in languages other than English, and position papers.

Notably, we also exclude papers where the primary contribution is not the application of a generative LLM for refactoring. This includes studies that utilize encoder-only models like BERT for classification tasks. For instance, we exclude works like Palit *et al.* [32], which uses GraphCodeBERT, a BERT-based model not designed for sequence to sequence generation, to identify refactoring candidates, and Alazba *et al.* [33], which focuses solely on code smell detection, not implementing any actual refactoring. This deliberate scoping decision is grounded in a fundamental methodological distinction. While studies like Liu *et al.* [34] demonstrate the effective application of BERT-based models to classification-specific tasks, like predicting a new identifier name, these models operate under a different approach than the generative LLMs that are the focus of this review. Encoder models like BERT are typically applied to refactoring by presenting it as a classification or simple prediction task (e.g., predicting a single token for a new method name), which is naturally evaluated with metrics like Accuracy and F1-score. For example, the SmellyBot tool [33] uses CodeBERT, a BERT-based model not designed for sequence-to-sequence generation,

to achieve high performance (e.g., 95.93% Accuracy, 96.82% Precision) on the specific, classification-based task of code smell detection. The tool operates by analyzing code to classify it into predefined categories of smells, functioning as a classifier rather than a generator, and does not implement any actual refactoring. In contrast, the generative LLMs covered in our primary studies, such as ChatGPT and LLaMA, are designed for open-ended sequence generation, which is essential for producing the diverse and complex code transformations inherent in most refactoring operations. The subset of the primary studies that reported metrics such as Precision and Recall to evaluate generative LLMs, applied them to the evaluation of their generated output as a whole (e.g., comparing an entire generated code snippet to a ground truth version), rather than assessing a single classification decision. Therefore, to maintain a consistent analytical framework for synthesizing results, particularly for RQ_4 (accuracy of suggestions) and RQ_5 (refactoring characteristics), we limit our primary studies to those investigating this distinct class of generative models. It is important to note, however, that several of the included primary studies do use static analysis tools or BERT-based models as baselines for comparison against their proposed generative LLM approaches. Where these comparative results are reported within our selected primary studies, we synthesize and discuss them to provide insight into the relative performance of generative LLMs against these alternative methodologies.

Stage 1: Identification of potentially relevant articles. We started the selection process by conducting an initial search across our databases, applying our search string to titles, abstracts, and keywords based on each library's search capabilities. This search yielded a total of 867 research publications. The text-string search's results were unrestricted, ensuring an extensive collection of potentially relevant studies.

Stage 2: Removal of duplicates. After merging all the search results, we identified and removed duplicate publications, which reduced the dataset to 558 unique research papers and eliminated 309 articles.

Stage 3: Exclusion of articles based on title and abstract. To narrow down our selection, we applied our inclusion and exclusion criteria to the titles and abstracts of the remaining studies. This process narrowed down our dataset to 79 papers, and when we could not determine relevance from just the title and abstract, the study was kept for further analysis.

Stage 4: Exclusion of articles based on full text. The remaining publications were fully analyzed to assess their relevance, and research that did not align with our inclusion and exclusion criteria was filtered out. As a result, 50 papers were selected for our study.

3.3. Study Quality Assessment

To assess the quality of PSs, we followed the guidelines proposed in [35–37]. We selected three quality assessment questions that could apply to all PSs. These questions serve to evaluate the study quality of each PS in the following three dimensions: objective, method, and coverage of the studies. The questions are as follows: Q1) *Does the study's primary objective explicitly focus on*

the LLM-based refactoring?; Q2) *Does the study include automatic or semi-automatic LLM-based refactoring approaches?*; Q3) *Does the study sufficiently describe the LLM-based technique, underlying algorithms, and evaluation methods?*. These questions are implicitly used in the above filtering process. If a PS passes these quality criteria, we believe that it has valuable information for SLR. The answer to each of these questions is either "Yes", "Partially", or "No" with numerical values of 1, 0.5, or 0, respectively. If the questions did not apply to the context of a PS, they were not evaluated. The overall quality of each PS is calculated by summing up the scores of the applicable questions. Overall, all the published articles in the accepted literature scored well on the quality assessment questions.

3.4. Data Extraction, Categorization, and Analysis

To seek the gaps of current studies in LLM refactorings, we screened a total of 50 different papers on such topic.

After collecting relevant papers we created a spreadsheet categorizing important key attributes we wanted to point out in the papers. Within our spreadsheet we had attributes such as title, link to the paper, year it was published, authors, venue, data source, research questions addressed, topics investigated, tools used, type of analysis, LLMs used, refactoring types used, programming language examined, prompting techniques, final prompt, results, and extra notes for miscellaneous items that we wanted to point out. We had taken turns in reviewing the papers. There are two authors involved in reviewing and cross-checking the manual analysis of the research papers. In our process, after one author reviewed a paper, the other would circle back and review it as well, then check the other's notes to ensure correctness.

After reviewing the key points of the articles, we analyzed each article again. For our second viewing of the papers, we set forth to write a summary of each of the papers. The summaries are broken down into three specific parts: goal, approach, and results. This helped us get a clear reminder of each paper when we needed a refresher while creating this paper, especially when writing RQ_1, RQ_3, and RQ_4. After the initial summary, we took note of the methodology of the research paper, specifically, how the data was collected, and what the authors did to collect the data. This can be anything ranged from how many datasets were collected, to how the data was preprocessed and processed, what the refactoring process (RQ_1, RQ_5) was, the prompting process (RQ_3), filtering processes (RQ_2), any specific methods used (RQ_1), the kinds of data collected (RQ_2), any tools that were used (RQ_1), and models used (RQ_2). These small attributes that made up the various dataset points gave us an insight into some of the most popular and effective usages of LLMs for refactoring, along with giving us snippets of possibilities of what may be missing or lacking in terms of research and understanding. Then, after collecting the methodologies of the papers, we also collected the evaluations from the papers. Evaluations include the qualitative analysis and quantitative analysis, along with extra notes of interest (RQ_2). We took note of quantitative analysis by taking a look at the numbers provided, what was calculated (RQ_2), important metrics and data, such as

the number of tests (RQ_2), the percentage of success (RQ_4), and comparisons with previous data and tools (RQ_2). Qualitative analysis depended more on the overarching non-numeric information from the papers, such as effectiveness (RQ_4), baseline comparisons (RQ_2), surveys (RQ_2), manual comparisons (RQ_2) and categorizations (RQ_3, RQ_1). To ensure complete and correct collections of data, we also asked ChatGPT-4 to check over our written evaluations and methodologies to see if there are important data points missed and added to our notes. This process included sending ChatGPT-4 all of the written notes of summaries with the original papers, asking if there are any more notes or topics that should be included in the summary. This usually resulted in improved grammar, added with some sentences for extra details. The authors also reviewed each other's notes at the end of each iteration and met to communicate and come to a consensus whenever there were disagreements.

Once all the data and information had been collected and studied, we found that there are specific methods of approaches to LLM refactoring, these included Prompt Engineering, External Frameworks and Tools, and Refactoring Types. These are different approaches that we see resurfacing across multiple papers repeatedly. These categories have been further analyzed in RQ_1 and RQ_3 of this study.

In RQ_3 of the study, we studied the specific prompting techniques that were used in the papers, several of which were repeatedly mentioned and analyzed across multiple papers. Throughout our time collecting data and writing summaries of the primary studies, we developed a classification of the key challenges (RQ_6) encountered when automating refactoring using LLMs. We began the categorization process by comprehensively reviewing the spreadsheet and study summaries, listing the difficulties that were reported when refactoring with LLMs. Then, we grouped studies with similar limitations (e.g., code hallucinations, intent misinterpretation). Finally, we analyzed these groups to identify and write clear descriptions of each main challenge, as originally documented by the authors of the primary studies.

3.5. Final Primary Studies Selection

The selection process outlined in the previous section resulted in 50 relevant publications. The primary sources of these studies are summarized in Table 1, spanning a total of 6 different databases published at various journals, conferences, and workshops. The earliest relevant study was published in 2023, while the most recent appeared in 2025, highlighting an increasing research interest in the use of LLMs for refactoring and emphasizing the need for further research in this field.

4. Experimental Results

4.1. RQ_1: What are the tools used in LLM-driven refactoring?

Motivation. Due to the nature of progressing code bases and developments, it has become increasingly hard for developers to create and update code that

Table 1: Publication venues.

Venue	Publication Venue	PSs
Journal	Expert Systems with Application	[16]
Journal	Automated Software Engineering	[38]
Journal	IEEE Transactions on Software Engineering	[39]
Journal	Electronics	[40]
Journal	Information Sciences	[41]
Journal	Software and Systems Modeling	[42]
Journal	Journal of Systems and Software	[43]
Journal	ACM Transactions on Software Engineering and Methodology	[44]
Journal	International Journal of Software & Informatics	[45]
Journal	Future Internet	[46]
Journal	ACM Transactions on Design Automation of Electronic Systems	[47]
Journal	Technology audit and production reserves	[48]
Conference	International Conference on Mining Software Repositories	[17, 18, 49, 50]
Conference	International Conference on Automated Software Engineering	[51–55]
Conference	International Conference on the Foundations of Software Engineering	[5, 7, 9]
Conference	International Conference on Software Maintenance and Evolution	[6, 13]|[56]
Conference	Asia-Pacific Software Engineering Conference	[15]
Conference	International Symposium on Search-Based Software Engineering	[57]
Conference	Generative AI for Effective Software Development	[58]
Conference	Technical Symposium on Computer Science Education	[59]
Conference	International Conference on Service-Oriented Computing	[60]
Conference	International conference on AI Foundation Models and Software Engineering	[10]
Conference	International Conference on Program Comprehension	[61]
Conference	Brazilian Symposium on Software Quality	[62]
Conference	International Symposium on Software Testing and Analysis	[63]
Conference	International Conference on Computing and Machine Intelligence	[64]
Conference	International Conference on Software Engineering	[65]|[66, 67]
Conference	International Conference on Evaluation and Assessment in Software Engineering	[11]
Conference	SoutheastCon	[68]
Conference	International Conference on Collaborative Advances in Software and Computing	[69]
Conference	International Conference on Software Analysis, Evolution and Reengineering	[70]
Conference	International Conference on Software Architecture Companion	[71]
Conference	International Joint Conference on Conceptual Knowledge Structures	[12]
Conference	Simpósio Brasileiro de Engenharia de Software	[72, 73]
Conference	Simpósio Brasileiro de Banco de Dados	[74]
Workshop	NeurIPS 2024 Workshop on Open-World Agents	[75]

maintains readability, maintainability, and efficiency [3]. One way to aid in this challenge is by deploying the power of LLMs, but studies vary in their use of tools to compare or aid LLMs in refactoring, so this section is primarily discussing the variety of tools used, their application in the study, and the role they play in the LLM-based system.

Approach. To address these issues, multiple studies have explored LLM-driven refactoring approaches that involve more effective tools with the LLMs. We reported some of these studies' characteristics in Table 2, giving a quick view of all of the papers regarding what tools, LLMs, and programming languages they used. The table differentiates between pretrained models based on their architecture and functional role. Models that include decoder components like CodeT5, PLBART, CodeGPT-adapt, CodeGen, and CodeGPT can be considered as either tools or LLMs based on their purpose. When used within

a larger LLM system, they function as supporting tools to an LLM, but when they independently generate or refactor code and are used without a supporting larger LLM, they are regarded as LLMs. By contrast, encoder-only models like BERT and its variants lack generative capability and are therefore classified as tools, as discussed earlier.

Results. We found that the use of tools provided valuable insight for improving LLM refactoring systems and acted as benchmarks to measure LLMs' refactoring ability. Here is the breakdown of the different tools and how they were used.

External frameworks and tools are defined as extra helpers used alongside LLMs to enhance their accuracy and efficiency [5–10, 39, 42, 50, 53, 54, 65]. Models and tools used in code smell detection have been found to be highly effective in working together with LLMs to implement higher accuracy in detection and automated refactoring. One such tool that was studied and examined was the iSMELL model which used CodeBERT to encode deep semantic information from code snippets into a vector capturing its multifaceted features. When using the iSMELL model combined with LLMs, it is found to achieve a 28.7% improvement over standalone LLMs [54]. Additionally, in multiple papers, pretrained code models were implemented to encode code into vectors used as node features in hypergraph neural networks to be provided to an LLM to validate the generated refactoring suggestions to implement Extract Method and Move Method refactoring [51, 52, 63]. A hybrid approach combining structured LLM guidance with rule-based techniques for Python refactoring has achieved over 90% Accuracy, demonstrating that structured LLM-driven methodologies can significantly enhance the reliability of code refactoring [39]. One paper used IDE-based static analysis to filter out unrelated classes and VoyageAI to measure semantic similarity between source class and candidate target class to improve Move Method and effectively eliminate all hallucinations that previously affected 80% of generated results [56]. Palit and Sharma [11] integrated reinforcement learning into their approach to improve LLM refactoring output from 41 successful tests for the CodeT5 model to 66 successful tests when implementing reinforcement learning. The RefactoringMiner tool was used to identify instances of refactoring to generate datasets and provide the LLMs with examples in multiple papers [11, 29, 38, 39, 56].

Plugins are also shown to be effective in improving current performance in refactoring by LLMs. One such plugin that was studied was EM-Assist [5, 6], a tool that integrates LLMs with IntelliJ IDEA, and aids in using ranking mechanisms and validation techniques to refine generated refactoring suggestions. This approach of improving generation was compared with traditional static analysis-based tools such as JExtract, and has been found to create an improvement in Recall from 39.4% to 53.4% [5, 6]. LLMs were also effective in testing existing refactoring engines in IntelliJ IDEA, Eclipse, NetBeans, and VSCode-Java, improving refactoring bug detection by identifying 86 bugs compared to the baseline approach of 24 bugs [65]. Across all fields of metrics, the impact of these tools, frameworks, plugins, and approaches has been largely positive.

Multiple studies utilized refactoring tools as benchmarks to compare LLM re-

factoring approaches. CodeReviewer was a popular benchmark when discussing code reviewing tasks with 4 papers mentioning it [45, 66, 67, 70], and refactoring tools like JDeodorant and JMove, both Java refactoring tool, were compared to multiple LLM refactoring approaches. [41, 51, 52, 54, 56, 63]. The results of these comparisons are further explored in RQ_4 and RQ_5.

Table 2: Related work in LLM-driven refactoring (RQ_1).

Study	Year	Focus	Tools/Models	LLMs	Language
Gehring [13]	2023	Introduces OpenRewrite, a tool for large-scale code refactoring.	Moderne, OpenRewrite, SonarQube	StarChat, ChatGPT-3.5	Java, Python, JavaScript, COBOL
White *et al.* [58]	2023	Prompt engineering for code refactoring	IntelliJ IDEA	ChatGPT-3.5 Turbo, ChatGPT-4	Python
Shirafuji *et al.* [15]	2023	Effectiveness of few shot prompting	Aizu online judge, YAPF, Radon, Python tokenizer	ChatGPT-3.5	Python
Depalma *et al.* [16]	2024	Correctness/Behavior Preservation/Documentation	PMD	ChatGPT-3.5	Java
AlOmar *et al.* [17]	2024	Refactoring on documentations, quality, and software repositories	Hacker News, SQLite, FastText, Google Translator, NLTK and Spacy, Dask	ChatGPT	Python, Java, JavaScript, C++, C#, TypeScript, Go, Ruby
Charan *et al.* [18]	2024	Examines how developers interact with ChatGPT, how they prompt it to refactor code, and how many prompts they make on average before reaching a conclusion	Hacker News, Google Translator	ChatGPT	Python, JavaScript
Pomian *et al.* [5]	2024	The effectiveness of EM-Assist in applying refactoring, especially the extract method	EM-Assist (developed plug-in tool), IntelliJ IDEA APIs, JExtract	GPT-3.5 Turbo	Java, Kotlin
Pomian *et al.* [6]	2024	The creation and need for EM-Assist, the inspection of how we can create a new refactoring tool by harnessing the power of LLMs	EM-Assist (developed plug-in tool), IntelliJ IDEA APIs, JExtract, Kolin and Java codebases	ChatGPT-3.5, ChatGPT-4, PaLM	Java, Kotlin
Dilhara *et al.* [7]	2024	How effective LLM is towards different ways of prompting for refactoring	PyCraft, R-CPATMiner, PyEvolve, Mutmut	ChatGPT-3.5, ChatGPT-4, PaLM	Python
Gao *et al.* [9]	2024	How can adding context using the approach of UTRefactor improve the refactoring outputs from LLMs	GitHub, Maven, tsDetect, UTRefactor, TESTAXE, JUnit 5, Llama	ChatGPT-4o	Java
Zhang *et al.* [39]	2024	Refactoring of non-idiomatic python code to pythonic idioms using hybrid approach that combines rule-based systems and LLMs; addresses code detection, correctness, idiomatic transformation, evaluates the proposed method against benchmarks and explores its scalabilities	ARIs, RIdiom	ChatGPT-3.5 Turbo	Python
Choi *et al.* [57]	2024	Testing how much a LLM-based refactoring pipeline effects the Cyclomatic Complexity of various Java projects, measured with the Lizard library	EvoSuite ,Defects4J, Lizard	ChatGPT-3.5 Turbo	Java
Wu *et al.* [54]	2024	Testing if combining the output of code smell detection tool sets recommended by Mixture of Experts(machine learning model) with LLMs will improve their code smell detection capabilities	PMD, JDeodorant, Organic, DesigniteJava, JSpIRIT, FeTruth, JMove, CodeBERT, CKmetrics	ChatGPT-3.5 turbo, ChatGPT-4.0 turbo, LLAMA3-70B, CodeLlama-34B	Java
Ishimw *et al.* [59]	2024	Code repair/refactoring, Automated feedback systems	Refractory, Google Cloud Platform's Google Compute Engine	ChatGPT-3.5 Turbo, ChatGPT-4	Python
Gautam *et al.* [75]	2024	Testing a benchmark for multi-file refactoring tasks, State-ful reasoning	SWE-bench	ChatGPT-4 Turbo, Claude 3.5 Sonnet	Python
Baumgartner *et al.* [40]	2024	State-aware behavior of LMMs. LLMs(ChatGPT) are used to generate tasks to test the LMMs on Testing different techniques of implementing LLMs(ChatGPT) into a refactoring workflow/pipeline using prompt engineering. These include data chunp detection, and refactoring tasks	ArgoUml	ChatGPT-4 Turbo, ChatGPT-3.5	Java
Cui *et al.* [51]	2024	Move Method refactoring	Graph Neural Network models: GraphSAGE, GAT, HGCN, HGNN, GCN, Refactoring tools: LLM Refactor, FeTruth, FeDeep, PathMove, JDeodorant, JMove, RMove, FeGNN, and FePM	Code Llama, DeepSeek, ChatGPT-4, ChatGPT-3.5	Java
Zhang *et al.* [41]	2024	Move Method refactoring	Deep Learning Models: CNN, LSTM, GRU, Natural language processing models: RoBERTa, ALBERT, XLNet, Refactoring tools: PathMove, JDeodorant, JMove, RMove,Pydriller, CodeSplitJava, RefactoringMiner, iPlasma, PMD, Cayenne, Pinpoint	ChatGPT-3.5 Turbo	Java
Ben Mrad *et al.* [60]	2024	Programming/Method Nomenclature Prediction and Automation	Nvidia T4 GPU	ChatGPT-3.5, Llama2-6.74B, Mistral-instruct-v0.2-7.24B	Java
Wang *et al.* [63]	2024	Introduces the design and uses of HECS, a tool that gives extract class refactoring suggestions	Refactoring Tools: JDeodorant, SSECS, LLMRefactor, Graph Neural Networks: HGNN, SAGE, GCN, GAT, Communities: GitHub, F-Droid, and Apach, Code Models: CodeBERT, CodeT ,CodeGPT, GraphCodeBERT, PLBART, CoTexT	ChatGPT-3.5	Java
Gao *et al.* [53]	2024	The effectiveness of RRG (Retrieve, Refactor, Generate)	Javalang, Python AST	ChatGPT-2, ChatGPT, CodeGEN, PolyCoder, CodeT5	Java, Python
Cui *et al.* [52]	2024	Explores the usefulness of a new tool for extract class refactoring suggestions. It combines hypergraph learning with an LLM	Refactoring Tools: JDeodorant , SSECS, LLMRefactor, Graph Neural Networks: HGNN, SAGE, GCN, GAT, Communities: GitHub, F-Droid, and Apache, Code Models: CodeBERT, CodeT ,CodeGPT, GraphCodeBERT, PLBART, CoTexT	GPT-3.5	Java
Zhang *et al.* [55]	2024	The code smells/quality of the code generated by copilot, effectiveness of copilot in fixing code	The code smells/quality of the code generated by copilot, effectiveness of copilot in fixing code	GitHub Copilot (derived from OpenAI Codex)	Python
Menolli *et al.* [62]	2024	Evaluating the effectiveness of a refactoring teaching method	Refractory	ChatGPT-3.5	Java
Wang *et al.* [10]	2025	The reliability and testing of refactoring engines in IDEs such as IntelliJ and Eclipse. uses RETESTER that use historical bug reports and LLMs to generate test inputs. automated detection of bugs in refactoring engines, evaluates the effectiveness of different input program characteristics	JVM Compiler, Eclipse's Bugzilla, GitHub issue trackers, and IntelliJ IDEA's YouTrack, Eclipse, IntelliJ IDEA, ASTGen, SAFEREFACTOR	ChatGPT-4	Java
Midolo *et al.* [61]	2025	The effectiveness of ChatGPT-4 in recommending and suggesting idiomatic refactoring actions. automated refactoring	RIdiom, GitHub	ChatGPT-4	Python
Ksontini *et al.* [50]	2025	The dockerfile quality checking, refactoring capabilities, automated vs manual, CI/CD pipelines	DRMiner, PARFUM, BM-25, Google BigQuery Github, automated build system	ChatGPT-4o	Docker, Python, bash, SQL
Alturayeif & Hassine [42]	2025	Decreasing bad smells when using GORE, types of linguistic bad smells in goal models	Stanza, BERT, jUCMNav	ChatGPT-3.5 turbo	Python (but the focus is on english)
Divyansh *et al.* [64]	2025	Effectiveness of ChatGPT on the cyclomatic complexity, cognitive complexity, code smells, and time debt after refactoring	SonarQube, Node, Stepik and Hyperskill	ChatGPT-3.5	Python, Javascript(for dataset extraction)
Liu *et al.* [38]	2024	The potentials of LLMs in automated software refactoring	ReExtractor, RefactoringMiner, IntelliJ IDEA, Dice Coefficient, Point-Biserial Correlation Coefficient	ChatGPT-4, Gemini	Java
Guo *et al.* [66]	2024	Investigate ChatGPT's ability to perform automated code refinement based on code review comments	CodeReviewer	GPT-3.5 Turbo, GPT-4	C, C++, C#, Go, Java, JavaScript, PHP, Python, Swift, Objective-C, Kotlin, SQL, Perl, Scala, R
Dong *et al.* [65]	2025	The use of ChatGPT to test refactoring engines	IntelliJ, Eclipse, NetBeans, VSCode-Java	ChatGPT	Java
Ghammam *et al.* [49]	2025	Current refactoring practices and technical debt in build systems	Maven, Ant, Gradle	ChatGPT-4o	30 languages but mainly: Java, Kotlin, PHP
Batole *et al.* [56]	2025	Implementing move method refactoring using LLMs	JMove, FeTruth, RefactoringMiner, VoyageAI	GPT-4o	Java
Palit & Sharma [11]	2025	Combining supervised fine-tuning with reinforcement learning for automated extract method refactoring	SEARTtool, RefactoringMiner, tree-sitter	Code-T5, PLBART, CodeGPT-adapt, CodeGen	Java
Rajendran *et al.* [68]	2025	Specializing LLM-based multiple agents based on different needs of refactoring	Not Specified	ChatGPT-4	Java
Kim [43]	2025	Comparing the validity of results from LLMs identifying relevant design patterns and then applying them to a location in a program	Not Specified	ChatGPT-4o, Claude 3.5 Sonnet, Microsoft 365 Copilot, Gemini 1.5, Llama 3.1-405B	Java
Wang *et al.* [44]	2025	Refactoring engine bugs	Eclipse, IntelliJ IDEA, NetBeans, SAFEREFACTOR-based tools, ASTGen	ChatGPT-4o mini	Java
Wang *et al.* [45]	2025	Evaluate the performance of LLMs vs pre-trained models in code refinement tasks and proposes a technique to mitigate the disadvantages of LLMs	LitGPT	CodeLLaMA, StableCode3B, LLaMA3, and ChatGPT-4o, T5-Review, CodeT5, CodeReviewer, and CodeBERT	C, C++, C#, Go, Java, JavaScript, PHP, Python, Ruby
AlOmar *et al.* [69]	2025	Investigate refactoring conversations between developers and ChatGPT	SQLite, FastText, Google Translator, NLTK, Spacy, Dask	ChatGPT-4, ChatGPT-3.5	Python, JavaScript, Bash
Midolo *et al.* [46]	2025	ChatGPT's abilities in refactoring for-loops into streams	JavaParser, Apache Maven (seen in code)	ChatGPT-3.5 Turbo	Java
Pucho *et al.* [73]	2025	Automated Python code refactoring using LLM-based multi-agent systems in machine learning projects	MetaGPT, PyRef, PyDriller, Radon, Lizard	Mixtral-8x22b	Python
Caumartin *et al.* [70]	2025	Comparing small open-source models to ChatGPT in code refinement tasks	CodeReviewer	Llama 2 7b, CodeLlama, GPT-3.5 Turbo	C, C++, C#, Go, Java, JavaScript, PHP, Python, Swift, Objective-C, Kotlin, SQL, Perl, Scala, R
Pandini *et al.* [71]	2025	The use of LLMs to suggest refactoring for Cyclic Dependency architectural smells	Arcan, LangChain, Ollama, Amazon Bedrock	Claude 3.5 Sonnet, Qwen	Java
Guo *et al.* [67]	2025	Automated code refinement to generate revised code based on identified intentions	CodeReviewer, T5CR, GitHub REST API, BM25	GPT-3.5, GPT-4o, DeepSeekV2, DeepSeek7B, CodeQwen7B	C, C++, C#, Go, Java, JavaScript, PHP, Python
Xu *et al.* [47]	2025	Automation of refactoring and optimization of C/C++ code for High-Level Synthesis	Catapult HLS Tool, Synopsys Design Compiler, Sentence Transformer	GPT-4 Turbo	C/C++
Tkachuk [48]	2025	Investigate the capability of LLMs to refactor code that improves maintainability and efficiency	Not Specified	ChatGPT, Copilot, Gemini, Claude	Java, Swift
Guenoune *et al.* [12]	2025	Combining Formal Concept Analysis (FCA), Relational Concept Analysis (RCA), and LLMs to help access the relevancy of derived concepts and name software entities appropriately	RCAtool, PlantText modeling tool	Gemini pro 2.5, Grok 3 thinking, ChatGPT o1, ChatGPT o3, DeepSeek V3, Claude 3.7 thinking 2, Claude 3.7 Somnet no think, Phind 2.0 70B	UML model that can be used to generate Python, C++, Java
Amaral [72]	2025	Investigate the potential of LLMs to automatically refactor JavaScript test smells	GitHub Copilot Chat, Amazon CodeWhisperer, SNUTS.JS, Steel, TestSmellDetector.Js, VSCode	GPT-4o (powers Github Copilot Chat), Amazon CodeWhisperer	JavaScript
De Sousa *et al.* [74]	2025	Evaluate LLMs and small language models ability in detection issues and bad smells in PL/SQL	Not Specified	GPT-4o, Gemini 2.0 Flash, GPT-4o mini, Phi-4	PL/SQL

Summary for RQ_1. *Tools are often used in the studies alongside LLMs to help bettering the performances of LLM refactoring and to collect data of the performances. There are external frameworks and tools used to aid LLMs for identifying code smells, generating data sets, ease of plugin, and measuring effectiveness. We have collected and categorized the tools used along with the LLMs, the languages used, and the focus of the specific study, giving an overview of the tools used in LLM-driven refactorings.*

4.2. RQ_2: What datasets and benchmarks have been used to evaluate LLMs for refactoring tasks?

Motivation. Harnessing the capabilities of LLMs has the potential to automate the costly and time-consuming task of refactoring in software engineering. Automating this process with LLMs could significantly reduce developer effort, minimize human error, and enhance software quality at a large scale. Numerous new LLM-based tools and approaches have been proposed in the wake of the widespread accessibility to these models in recent years. However, with each study relying on different datasets and benchmarks to evaluate these models, there is variation in how effectiveness is measured. In this section, our analysis focuses on examining the datasets and benchmarks referenced across the papers, looking at their source, composition, and language. This analysis allows us to better identify patterns in the datasets and sources of the papers, as well as further understand how these factors could bias the evaluations and ultimate conclusions reached about the refactoring capabilities of these LLMs. Therefore, it is important to analyze the data they use in their approaches and compare them to understand the nature of these evaluations.

Approach. To further understand the quality, workflow, and design of the experiments in each of the primary studies, we report in Table 3 the following three characteristics:

- *Source:* Indicates where the dataset or benchmark originated from.
- *Dataset:* Indicates the type and quantity of the data within each dataset or benchmark.
- *Language:* Indicates the programming language(s) represented in the dataset or benchmark.

We report the source of any dataset or benchmark from previously published research as 'Previous work' along with the link to that work. Additionally, the source of any datasets or benchmarks extracted from unnamed projects is labeled 'Various open-source'.

Results. In the following section, we discuss the sources, datasets, and languages used to evaluate LLMs for refactoring tasks.

Sources. There were a variety of sources used to evaluate LLMs for refactoring. Eleven out of the 50 primary studies, 22%, relied on previously published work and referenced established datasets to ensure the credibility of their evaluations.

However, we identified 35 primary studies (70%) that collected data from open-source projects, with code repositories hosted on platforms such as GitHub, Apache, and F-Droid. This trend toward data extraction from open-source projects for conducting these studies is due to the accessibility of this source code, as researchers can analyze real-world refactoring instances without restriction. Additionally, two studies (4%) sourced their datasets from educational environments, using student program submissions to test their refactoring approach. Finally, two studies (4%) generated new datasets by systematically prompting LLMs and integrating dataset creation into the evaluation process. The diversity in dataset sources highlights the absence of a standardized approach to collecting data for LLM-based refactoring research.

Datasets. There is significant variation in the composition of the datasets and benchmarks used to evaluate LLMs for refactoring tasks, reflecting the variety of sources and overall diversity in approaches to evaluating the refactoring capabilities of LLMs. If the number of projects used is not explicitly mentioned in a study, we report the other quantities of data that are mentioned. Data types range from commits to full methods, and even include types of prompt patterns. The largest datasets contained up to 30,000 code samples and 41,311 refactoring cases, while the smallest contained only 116 methods. Moreover, some datasets [15, 59] contained full program submissions for multiple programming problems. This range highlights that while some evaluations focus more on scale, others prioritize depth and context. Consequently, datasets built from commits or real student submissions may provide more realistic scenarios, corresponding with the more common use cases of these LLMs, leading to different evaluation outcomes than those based on isolated methods or artificially constructed examples, which may test narrower and more basic capabilities of LLMs. This variation in dataset composition and sources again reinforces the lack of a standardized benchmark for evaluating LLM-based refactoring, meaning there is inconsistency in how model effectiveness is measured across primary studies.

Languages. The most common programming language contained in the datasets is Java, which is present in 33 (66%) of the primary studies [5, 6, 9–13, 16, 17, 38, 40, 41, 43–46, 48, 49, 51–54, 56, 57, 60, 62, 63, 65–67, 70, 71]. This is followed by Python [7, 12, 13, 15, 17, 18, 39, 42, 45, 50, 53, 55, 58, 59, 61, 64, 66, 67, 69, 70, 73, 75] which makes up 22 (44%) studies. The next two most prevalent languages were JavaScript [13, 17, 18, 45, 64, 66, 67, 69, 70, 72] and C++ [12, 17, 45, 47, 66, 67, 70], which appeared in ten (20%) and seven (14%) of the studies respectively. There are multiple instances of other languages including C#, Kotlin, SQL. However, they appear much less frequently than the aforementioned languages. For example, there are instances of Java and Kotlin [5, 6], Java and Python [53], Python and Javascript [18], Python, Bash and SQL [50], as well as Python, Java, Javascript, C++, C#, Typescript, Go and Ruby together. Additionally, Ghammam *et al.* used mainly Java, Kotlin, and PHP, but incorporated 27 other languages [17, 49]. There was a study [74] that exclusively used languages other than Java, Python, JavaScript, and C++, making up 2% of the total of 50 primary articles. The dominant trend of Java and Python data is due to the prevalence of the languages' within exist-

ing datasets and benchmarks in software engineering research, refactoring tools, university curricula, and personal and industry projects. The variety of programming languages indicates that software engineering research is expanding to explore refactoring of queries and scripts, although it is still mainly skewed towards traditional codebase refactoring.

Data Leakage. When it comes to preventing data leakage, there are a few common methods for preventing data leakage when evaluation of language models. The ones considered in Table 3 are training/testing splits, unseen projects or repositories, held-out datasets, and cross-validation. A training/testing split means dividing the datasets into separate sections for training and for evaluation, so that the model is not tested on data it has already seen. Unseen projects or repositories go a step further by evaluating models on entirely different sources than those used when training a model. A held-out dataset is a collection of data reserved only for testing that was never used during training. Finally, cross-validation splits the data into multiple folds (smaller subsets) and then cycles through multiple rounds where the model is tested on one fold and trained on the remaining folds, rotating the test fold each round.

A constant focus when it comes to an experiment is eliminating as much bias as possible. The critical need for these precautions is underscored by research on LLM threats within software engineering. As highlighted by Sallou *et al.* [76], a primary concern is implicit data leakage due to pre-training. LLMs are trained on vast public datasets like GitHub, which may include the very benchmarks (e.g., Defects4J) used for evaluating their capabilities. This can lead to models memorizing existing solutions rather than generating new ones, severely threatening the validity of research findings.

In the context of LLM-based refactoring, data leakage refers to language models being tested on the same data that was used to train them. For the most exact, unbiased study of LLMs, precautions such as held-out datasets, training/testing splits, cross-validation, etc., should be taken to prevent data leakage. In our analysis of the various datasets and benchmarks used to evaluate LLMs on refactoring tasks, shown in Table 3, we found a widespread lack or omission of data leakage precautions by the authors. Overall, a large majority—36 out of 50 primary papers or 72%—made no mention of deliberately implementing data leakage precautions when training and evaluating their LLMs. Of the 14 papers that did specify taking such precautions, six (12%) used training/testing splits [11, 41, 49–51, 70], with the rest using held-out datasets [42, 52, 67] at 6%, unseen projects or repositories [38, 39, 66, 75] at 8%, or cross-validation [54] at 2%. For the remaining five papers, the application of data leakage prevention measures was unclear, and they were thus reported as 'Not clear' in the table. While the mention of preventative measures in almost a third, 28%, of the reviewed primary papers is encouraging, this highlights a significant gap in current evaluation practices or reporting of data leakage precautions, which fails to adequately address the fundamental threats of data contamination inherent in LLM-based research.

Notably, this review does not confirm the presence or absence of data leakage in the analyzed studies. Rather, we report only whether authors explicitly

mentioned implementing any preventive measures and, when specified, what those measures were. The inclusion of such precautions in a paper does not guarantee complete mitigation of data leakage, as verifying that falls beyond the scope of this review.

Figure 2 shows the distribution of programming languages used in the primary studies to evaluate refactoring tools. Java, Python, JavaScript, and C++ emerged as the most prevalent languages. The reported percentages are non-exclusive; a single study utilizing multiple languages contributes to the count for each language used. As the data indicate, Java is the most popular language in LLM-based refactoring research, featured in 66% (33) of the primary studies. It is followed by Python, employed in 44% (22) of the studies. JavaScript, in 20% (10) of the studies, and C++, in 14% (7) of the studies, were also common. Additionally, 24% included other less frequently used languages besides the four mentioned above. This overview provides a detailed picture of the current programming language focus in the field, offering useful information that could guide future researchers within this domain concerning the diversity in their language selections.

Table 3: Dataset in LLM-driven refactoring (RQ_2).

Study	Projects	Dataset	Language	Manual / Automated	Year	Data leakage precautions
Gehring [13]	openrewrite/rewrite, Spring Cloud, Netflix	1 recipe, 4 files	Java	Yes/No	2023	No (not mentioned)
White *et al.* [58]	Previous work[14]	14 prompt patterns	Python	Yes/No	2023	No(not mentioned)
Shirafuji *et al.* [15]	Aizu Online Judge	8,000,000 submissions: 880 selected (avg. LOC: 14.82)	Python	Yes/Yes	2023	No(not mentioned)
Depalma *et al.* [16]	Previous work [77]	40 files: 1 class (LOC: 15 - 35)	Java	Yes/Yes	2023	No(not mentioned)
Alomar *et al.* [17]	DevGPT [78]	470 commits, 69 issues, 176 files	Python, Java, Javascript, C++, C#, TypeScript, Go, Ruby	Yes/Yes	2023	No(not mentioned)
Chavan *et al.* [18]	DevGPT [78]	447 conversations: 1,246 records	Python, Javascript	Yes/Yes	2023	No(not mentioned)
Pomian *et al.* [5]	Various open-source	1,756 refactorings: (avg. LOC: 30)	Java, Kotlin	No/Yes	2024	No(not mentioned)
Pomian *et al.* [6]	MyWebMart, SelfPlanner, WikiDev, JHotDraw, JUnit	122 methods	Java, Kotlin	Yes/Yes	2024	No(not mentioned)
Dilhara *et al.* [7]	Various open-source	9,325 code change patterns	Python	Yes/Yes	2023	No(not mentioned)
Gao *et al.* [9]	google/gson, jhy/jsoup, apache/commons, jfree/jfreechart	6 projects: 9,149 tests, 2,375 code smells (total LOC: 174,510)	Java	No/Yes	2024	No (not mentioned)
Zhang *et al.* [39]	LLM-generated ARIs	3,311 methods, 4678 code pairs	Python	No/Yes	2024	Yes: unseen projects
Choi *et al.* [57]	Defects4J	17 projects	Java	No/Yes	2023	Not clear
Wu *et al.* [54]	Previous work [79][80][81]	3 widely-used datasets	Java	No/Yes	2024	Yes: cross-validation
Ishizue *et al.* [59]	Large public university	361 students: 5 assignments: 1,800 submissions	Python	No/No	2023	No (not mentioned)
Gautam *et al.* [75]	Various open-source	9 projects: 100 multi-file refactoring tasks	Python	Yes/Yes	2024	Yes: unseen projects
Baumgartner *et al.* [40]	ArgoUml	LOC: 180,000	Java	Yes/No	2024	No(not mentioned)
Cui *et al.* [51]	GitHub, F-Droid, Apache	11,169 projects	Java	Yes/Yes	2024	Yes: train/test split
Zhang *et al.* [41]	Various open-source	58 projects: 639,010 total methods	Java	No/Yes	2024	Yes: train/test split
Ben Mrad *et al.* [60]	Previous work[82]	116 methods	Java	No/Yes	2024	No (not mentioned)
Wang *et al.* [63]	GitHub, Apache, F-Droid, GanttProject, Xerces	11,336 projects: 41,311 refactoring cases	Java	Yes/Yes	2024	No (not mentioned)
Gao *et al.* [53]	CodeSearchNet, Concode	966,629 code samples: 30,000 selected	Java, Python	Yes/Yes	2024	No (not mentioned)
Cui *et al.* [52]	GitHub, Apache, F-Droid, GanttProject, Xerces	11,336 projects: 41,311 refactoring cases	Java	No/Yes	2024	Yes: held-out projects
Zhang *et al.* [55]	GitHub Copilot	1,204 files	Python	Yes/Yes	2023	No (not mentioned)
Menolli *et al.* [62]	Unknown GitHub Projects	23 students selected 4 projects	Java	Yes/No	2024	No (not mentioned)
Wang *et al.* [10]	Eclipse, GitHub, IntelliJ IDEA	458 bug reports	Java	No/Yes	2025	No (not mentioned)
Midolo *et al.* [61]	idiomaticrefactoring/pythonidiomsrefactor	736 projects: 1,061 methods total	Python	Yes/No	2024	Not clear
Ksontini *et al.* [50]	Google BigQuery GitHub Public Dataset	600 dockerfiles	Python, Bash, SQL	Yes/Yes	2024	Yes: train/test split
Alturayeif & Hassine [42]	Goal-oriented Requirements Language	4 categories: 17 linguistic smells total	Python	Yes/No	2024	Yes: held-out datasets
Divyansh *et al.* [64]	Previous work [83]	1,032 files	Python	Yes/Yes	2024	No (not mentioned)
Liu *et al.* [38]	Various open-source	20 projects: 180 refactorings	Java	Yes/Yes	2024	Yes: unseen repositories
Guo *et al.* [66]	GitHub/CodeReview	467 projects, 14,568 samples	16 languages including: C, C++, Swift, Kotlin	Yes/Yes	2023	Yes: unseen repositories
Dong *et al.* [65]	Eclipse, IntelliJ IDEA, and VSCode	700 bug reports and 185 test cases	Java	Yes/Yes	2024	No (not mentioned)
Ghammam *et al.* [49]	GitHub	725 commits from 609 projects	30 languages but mainly: Java, Kotin, PHP	Yes/Yes	2024	Yes: train/test split
Batole *et al.* [56]	Various open-source	10 projects, 210 refactorings instances	Java	Yes/Yes	2024	No (not mentioned)
Palit & Sharma [11]	GitHub	1,618 projects, 33,477 samples	Java	Yes/Yes	2024	Yes: train/test split
Rajendran *et al.* [68]	Various open-source	4 agents: dataset composition not specified	Java	Yes/No	2024	No (not mentioned)
Kim [43]	Various open-source	2 programs: 11 and 6 classes	Java	Yes/No	2024	Not clear
Wang *et al.* [44]	Eclipse, GitHub, IntelliJ IDEA	1,651 bug reports	Java	Yes/Yes	2024	No (not mentioned)
Wang *et al.* [45]	GitHub and Gerrit	800 samples	9 languages including: C, C++, C#, Go, Java, JavaScript	Yes/No	2024	Not clear
AlOmar *et al.* [69]	DevGPT [78]	470 commits, 69 issues, and 176 files	Python, JavaScript, Bash	Yes/Yes	2024	No (not mentioned)
Midolo *et al.* [46]	GitHub	4 projects, 2,132 loops	Java	Yes/Yes	2025	Not clear
Pucho *et al.* [73]	Various open-source	2,000 files	Python	No/Yes	2024	No (not mentioned)
Caumartin *et al.* [70]	GitHub/CodeReview-New	467 projects, 14,568 samples	16 languages including: C, C++, Swift, Kotlin	No/Yes	2024	Yes: train/test split
Pandini *et al.* [71]	Keyword, Various open-source	5 projects	Java	No/Yes	2024	No (not mentioned)
Guo *et al.* [67]	GitHub/CodeReview	15 projects, 1,100 code reviews	C, C++, C#, Go, Java, JavaScript, PHP, and Python	Yes/Yes	2024	Yes: held-out datasets
Xu *et al.* [47]	Previous work[84]	74 files, 24 examples	C/C++	Yes/No	2024	No (not mentioned)
Tkachuk [48]	Unknown	4 code snippets	Swift, Java	Yes/No	2024	No (not mentioned)
Guenoune *et al.* [12]	Previous work[85]	2 projects	Python, C++, and Java are mentioned	Yes/No	2023	No (not mentioned)
Amaral [72]	GitHub	10 projects, 148 test smells	JavaScript	Yes/Yes	2024	No (not mentioned)
De Sousa *et al.* [74]	Various open-source	44 code snippets	PL/SQL	Yes/No	2025	No (not mentioned)

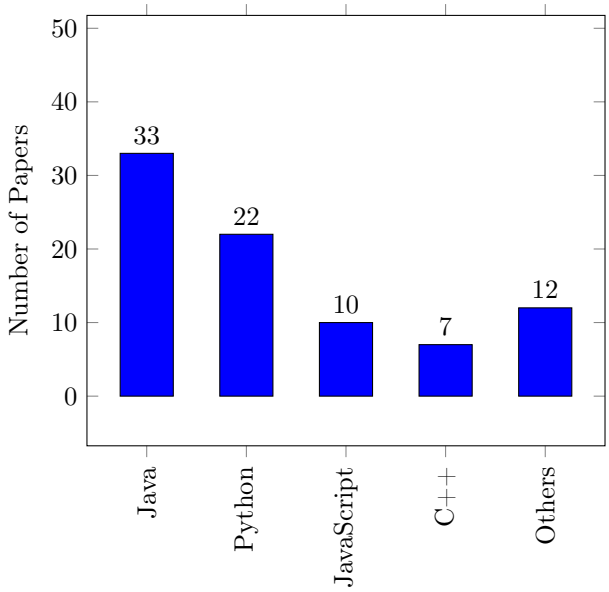

Figure 2: Programming languages across primary studies.

Summary for RQ$_2$. *The datasets and benchmarks used to evaluate LLMs for refactoring tasks vary greatly in both their composition and scale, making their standardization for more consistent comparison difficult. A majority of studies, 70%, sourced data from open-source repositories, with Java (66%) and Python (44%) dominating the evaluation landscape, reflecting their prevalence in software engineering research. Additionally, 70% of studies fail to report measures against data leakage, underscoring a substantial validity concern within this emerging field.*

4.3. RQ$_3$: What prompt techniques have been used to perform refactoring tasks?

Motivation. The need for studying different prompting techniques in LLM-driven code refactoring comes from the need to improve the effectiveness, reliability, and adaptability of AI-assisted software development. While LLMs have shown promise in automating refactoring tasks, their performance depends heavily on how prompts are structured and syntactically formed. Poorly designed prompts can lead to inaccurate refactoring, making it crucial to identify and refine effective prompt engineering strategies. It is seen through studies that different refactoring tasks require varying levels of context and reasoning. Simple refactoring methods, such as renaming or adjusting indentation, can often be handled with basic prompts. However, more complex refactoring methods, such as Extract Method, demand more structured, iterative, and context-based prompts to ensure correctness and maintainability.

Approach. To address these challenges, researchers have explored different prompting approaches to determine which methods yield the most accurate, efficient, and maintainable code transformations. We reported the prompting technique, final prompts used, along with the best methods used by each study in Tables 4 and 5.

Results. Prompt engineering is defined as the process of creating and formulating inquires to LLMs to guide it in order to retain the most efficient and accurate results, thus falling under the category of LLM based approach [5–7, 9, 10, 12–18, 38–40, 42–48, 50–57, 59–64, 66, 67, 69–75]. We have found that several studies utilized prompt engineering in their research to guide LLMs in producing structured and semantically correct refactoring. There are multiple forms of prompt engineering implemented across various studies, and improvements and changes in prompting have been shown to be more efficient in creating refactored code. Below are the seven categories of prompt techniques we've encountered, listed, explained, and illustrated in Figure 3.

- Zero-Shot [13, 15–17, 39, 40, 43, 48–50, 60–62, 64, 66, 67, 69, 70, 72, 73, 75]: Implemented in 19 studies (16%), and provides minimal context and asks the model to refactor code directly.

- One-Shot [14, 15, 17, 18, 40, 46, 49, 50, 54, 57, 59, 69]: Implemented in 12 studies (10%), and provides a single example to enhance the model's ability to generate and generalize across multiple different refactoring tasks.

- Few-shot [5–7, 10, 12, 14, 15, 17, 38–40, 42, 44, 45, 47, 50–53, 55, 57, 61, 63, 64, 67, 69, 75]: Implemented in 25 studies (21%), and provides multiple examples to enhance the model's ability to generalize across different refactoring tasks.

- Chain-of-Thought [7, 9, 10, 12–14, 42, 47, 55–57, 66, 67, 69–71, 74]: Implemented in 18 studies (15%), and provides a step-by-step process of reasoning to improve the LLM's decision making in refactoring.

- Context-Specific [6, 10, 12, 14, 18, 38, 39, 39, 42, 47, 51–54, 57, 59–63, 65–67, 69–71, 75]: Most commonly implemented with 27 studies (22%) testing it, and provides additional information on the codebase, such as usage and location of the code that should be fixed, along with quality attributes such as readability or maintainability refactoring goals to achieve

- Output Constraints [10, 14, 18, 38, 39, 42, 47, 51, 53, 59–61, 69, 74, 75]: Implemented in 15 studies (12%), and gives specifications of what not to do or what to do in terms of formatting or correctness.

- Ranking [5, 6]: Implemented in 2 papers (2%), and asks the LLM to return back multiple versions of the refactored code, and to also rank the versions from best to worst.

By applying these prompting techniques, studies have assessed the effectiveness of improving refactoring outcomes in various code refactorings. The evaluations focused on factors such as accuracy, maintainability, and retention of the original functionalities, comparing different prompting strategies to determine which yields the most reliable results. Out of all of these techniques, it is evident that One-Shot prompting is the most effective for achieving correctness in Java code refactoring, improving test pass rates compared to Zero-Shot by 6.15% and reducing code smells at a 3.52% higher rate. One-Shot prompting is also shown to have yielded the highest unit test pass rate of 34.51% and code smell reduction rate of 42.97% outperforming compared to both Zero-Shot and Chain-of-Thought prompting [8], and One-Shot prompting preforms better than Zero-Shot in built system refactoring receiving a Precision, Recall and F1-measure of 0.79, 0.78, and 0.76 compared to Zero-Shot's 0.67, 0.72, and 0.66 [49]. But in certain cases, such as Python-based refactoring tasks, it is shown that Few-Shot prompting performed the best, improving refactoring quality by reducing cyclomatic complexity (CC) by 17.35% and decreasing lines of code by 25.85% while maintaining correctness [15]. Additionally, when tested in multiple programming languages, Few-Shot prompts outperformed both Chain-of-Thought and Zero-Shot prompts [67]. Chain-of-Thought prompting is shown to improve diversity in refactoring, but in terms of correctness, it did not outperform One-Shot [8]. One study shows that context-enhanced prompting significantly reduces test smells, improving test code quality in automated test refactoring through the use of the UTRefactor framework. It was capable of reducing test smells from 2,375 to 265, achieving 89% smell elimination, compared to 55% for standard LLM-based approaches [9].

Summary for RQ_3. *There are various ways of prompt engineering when interacting with LLMs for refactoring, these types are: Zero-Shot, One-Shot, Few-Shot, Chain-of-Thought, Context-Specific, Output Constraints, and Ranking. We found that for different use cases in different studies, certain prompts are better than others. Which prompting technique to employ for prompt engineering thus become a highly situational dependent decision.*

4.4. RQ_4: How accurate are LLM-generated refactoring suggestions?

Motivation. LLMs offer the potential to automate refactoring, a time-consuming yet crucial task for maintaining software quality. Using LLMs to automate the refactoring process could significantly reduce development time and effort while enhancing the efficiency of countless software systems. Still, the accuracy of these suggestions is critical if they are to be integrated into existing systems, especially those managing sensitive information or impacting people's lives. The primary studies use different datasets, languages, and evaluation methods to measure the accuracy of their refactoring tools. Therefore, it is imperative to examine not just the accuracy of LLM-generated refactoring

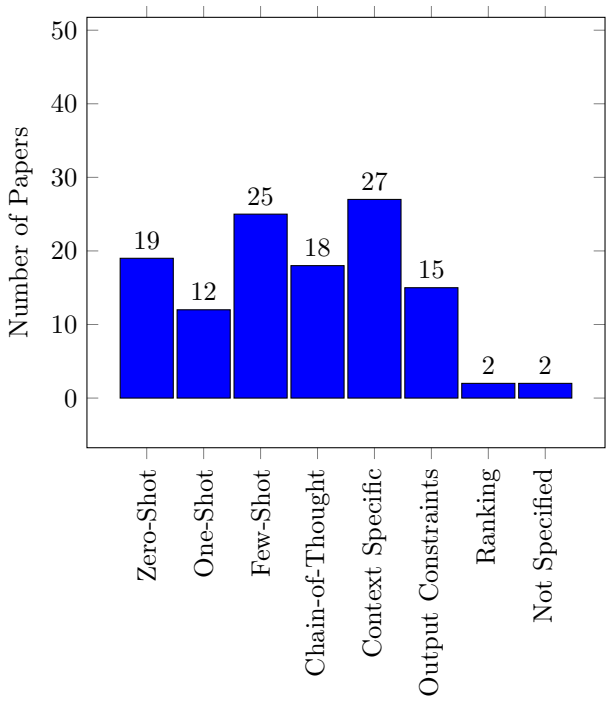

Figure 3: Prompt engineering techniques across primary studies.

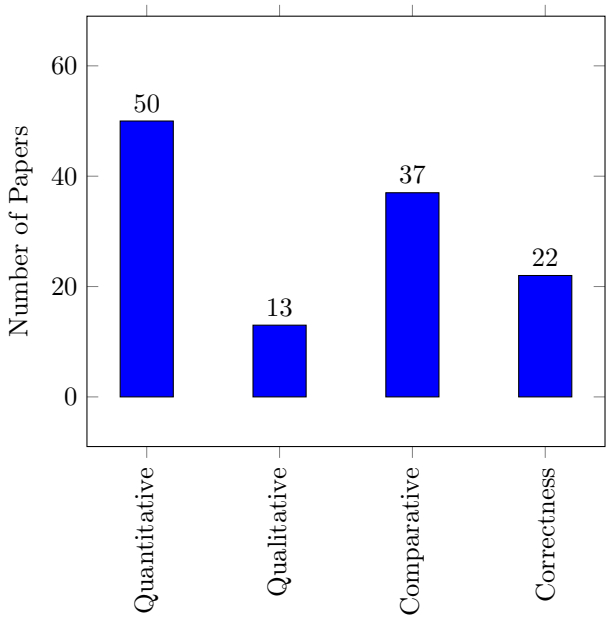

Figure 4: Quantitative, qualitative, and comparative analysis across primary studies.

Table 4: Prompt technique used in LLM-driven refactoring (RQ_3).

Study	Prompt Technique	Best Approach	Final Prompt
Gehring [13]	Zero-Shot, Chain-of-Thought	Using ChatGPT-3.5 with Chain-of-Thought, resulting 16 out of 20 repository results being correctly refactored	Repeat the code making sure that any default case is in the last case of a switch statement: [code]
White *et al.* [58]	Context-Specific, Chain-of-Thought, Output Constraints, Few-Shot, One-Shot	Using Context-specific prompting with Chain-of-Thought can best help improve LLM-generated refactoring	Multiple prompts are featured (see paper for individual prompts)
Shirafuji *et al.* [15]	Zero-Shot, One-Shot, Few-Shot	Using Few-Shot prompting leads to a 95.68% of successful refactoring	"Refactor the given Python program to a more readable, efficient, and maintainable one. You can assume that the given program is semantically correct. Do not change the external behavior of the program, and keep the syntactic and semantic correctness. Python programs should be in a code block. Do not explain anything in natural language."" provide snippets of code and ask for variety.
Depalma *et al.* [16]	Zero-Shot	Using Few-Shot prompting yielded best results from ChatGPT	"With no explanation refactor the Java code to improve its quality and [quality attribute]:"
			"Does this Java code preserve the behavior of the above code segment? Explain in detail why or why not:"
			"Provide concise commit messages that describe the overall goal, refactoring changes, and impact of quality of the following code: [include refactored code here]."
AlOmar *et al.* [17]	Zero-Shot, One-Shot, Few-Shot	Using Few-Shot prompting along with context-specific prompting yielded best results	No Final Prompt Provided
Chavan *et al.* [18]	Context-Specific, Zero-Shot, Output Constraints, One-Shot	No best approach was mentioned. The paper broke down how and what developers used LLMs for; there was two primary ways developers used ChatGPT for refactoring and it was via Chain-of-Thought prompting or One-Shot prompting	No Final Prompt Provided
Pomian *et al.* [5]	Context-Specific, Few-Shot, Ranking	The tool EM-Assist uses Few-Shot prompting provided with Context Specifes with and Rankings to recommend for best refactoring options	No Final Prompt Provided
Pomian *et al.* [6]	Context-Specific, Few-Shot, Ranking	The tool EM-Assist was able to successfully perform refactoring in 53.4% of the cases by using Few-Shot, Context-Specific, and Rankings	https://github.com/llm-refactoring/llm-refactoring-plugin/tree/main#prompt-example summary of the repo: You are an expert software developer skilled in Java and Kotlin refactoring. Your task is to identify and extract a method from the given long method below.... [Actual Code Here]
Dilhara *et al.* [7]	Few-Shot, Chain-of-Thought	Using Chain-of-Thought reasoning and few-shot prompting, the tool PyCraft was able provide input example and test cases that can identify help identify code change patterns successfully up to an overall success rate of 96.6%	Provide an snippet of the CPAT transformation example to set the context, then generate unit tests to verify correctness, set the temperature and iteration settings(T = 1.2, I = 5 for random; T = 0.5, 0.7, I = 3, I_f = 5 for structured generation), check for structural intent, and the filter out the incorrect variants
Gao *et al.* [9]	Chain-of-Thought	Using Chain-of-Thought, the tool UTRefactor is capable of achieving an 89% reduction of code smells, from 2,375 of the testing data to 265	"Role Assignment: ""You are a Java testing expert, and now you need to refactor the test code which contains bad test code smells."" Step-by-Step Reasoning Using CoT: Understand the intent of the test. Recognize the code smells present in the test code. Comprehend the refactoring rules and measures. Refactor the test code accordingly. Provided Information: Source code of the unit test. Context of the unit test (focal class, method, related functions). Detected smells and descriptions. Refactoring steps in DSL."
Zhang *et al.* [39]	Few-Shot, Context-Specific	Using Few-Shot and Context-Specific, the authors were able to achieve over 90% in improving python code idomization in accuracy and precision	No Final Prompt Provided
Choi *et al.* [57]	One-Shot, Few-Shot, Chain-of-Thought, Output Constraints, Context-Specific	Using Chain-of-Thought, LLMs was able to break complex methods to multiple simple methods, which allows the reduction of Cyclomatic Complexity to be reduced up to 10.4%.	"'''java Target Method ''' Refactor the provided Java method to enhance its readability and maintainability. You can assume that the given method is functionally correct. Ensure that you do not alter the external behavior of the method, maintaining both syntactic and semantic correctness. Provide the Java method within a code block. Avoid using natural language explanations."
Wu *et al.* [54]	Context-Specific, One-Shot	The tool iSMELL uses both One-Shot and Context-Specific methods and approaches to identify code smells, reaching an accuracy of 75.17%.	"<Instruction>: In computer programming, a code smell is any characteristic in the source code of a program that possibly indicates a deeper problem. I will now tell you the definition about <code smell>. Please read the definition and refactor the code according to the target to eliminate this smell. The definition of <code smell> is: <Definition>. Here is an example for refactoring code which has <code smell> smell, Please read it carefully and learn to show only the most modified critical code after refactoring. You can replace function code with comments. [code] Now based on the example, refactor the following code to eliminate the <code smell>.
Ishizue *et al.* [59]	Output Constraints, Context-Specific, One-Shot	Using Output Constraints, Context-Specific, One-Shot methods combined with Refactory, the authors are able to achieve near 88.8% - 99.5% of correct refactoring.	"This is a final prompt that will be provided if the previous prompt is unsuccessful(see paper pg. 571):..."
Gautam *et al.* [75]	Few-Shot, Output Constraints, Context-Specific	Using Few-Shot, Output Constraints, Context-Specific methods combined, the authors are able to reach 43.9% of the testcases resolved.colorblack	The prompt used on the SWE-agent is very long(see paper pg.17-18), there are also additional prompts used to create the instruction set(see pg. 16)
Baumgartner *et al.* [40]	Few-Shot, One-Shot, Zero-Shot	The study showed that the Few-Shot method is best used to find areas of refactoring.	No Final Prompt Provided
Cui *et al.* [51]	Few-Shot, Output Constraints, Context-Specific	Using Few-Shot, Output Constraints, and Context-Specific the authors were able to achieve an increase of 27.8% in precision, 2.5% in recall and 18.5% in f1 compared to other leading tools.	This is an example prompt: 1. There are now seven Pre-conditions for move method refactoring(P1-P7): 2. Consider the following test move method candidates(if the candidate move method is related to the Superclass, the corresponding Superclass is given below, otherwise it is empty): [Code] 3. Please verify whether the candidate meet pre-conditions P1-P7 by extracting the precondition relevant entries from the candidate and presenting the verification results in tabular form. (see paper for definitions of P1-P7)"
Zhang *et al.* [41]	Zero-Shot, Context-Specific, Output Constraints	By using Zero-Shot, Context-Specific, Output Constraints the tool MoveRec was able to improve the F1 from 9.4% to 53.4%.	You are a software engineer who is proficient in refactoring.
			You will do the work of code summary. Please make a concise summary based on the following code features. + [CODE]
Ben Mrad *et al.* [60]	Zero-Shot, Context-Specific, Output Constraints	By using Zero-Shot, Context-Specific, Output Constraints the tool MoveRec was able to improve the F1 from 9.4% to 53.4%.	No Final Prompt Provided
Wang *et al.* [63]	Few-Shot, Context-Specific	By using Few-Shot, Context-Specific, the tool HECS was able to produce a precision of 76.8%, recall of 84.3%, and F-1 of 80.4%.	No Final Prompt Provided
Gao *et al.* [53]	Context-specific, Few-Shot, Output Constraint	By using Context-specific, Few-Shot, and Output Constraints, the tool RRG was able to increase similarity of code generated and referenced by up to 28%.	No Final Prompt Provided
Cui *et al.* [52]	Few-Shot, Context-Specific	Using Few-Shot, Context-Specific, HECS was able to achieve 76.8% in precision, 84.3% in recall, and 80.4% in F1-Measure.	The P1-P8 preconditions are defined in detail in the paper. 1. Pre-condition rules (P1-P8) as follows: ... 2. Consider the following test extract class candidate: Source class:... Extract class candidate: - field: A.f2 - method: A.m2() Please verify whether the candidate meet pre-conditions P1-P8 by extracting the pre-condition relevant entities from the candidate and presenting the verification results in tabular form.
Zhang *et al.* [55]	Few-Shot, Chain-of-Thought	Using Few-Shot with Chain-of-Thought, Copilot Chat was able to achieve a successful refactoring rate of 87.1% in a code base of 102 code smells.	General Fix Prompt: Fix the problem in the selected code Code Smell Fix Prompt: Fix the code smell in the selected code Specific Code Smell Fix Prompt: Fix the [code smell name] (e.g., Long Method) code smell in the selected code
Menolli *et al.* [62]	Context-Specific Prompting, Zero-Shot Learning	No best prompt was mentioned, but by using Context-Specific prompting, the participants were able to use ChatGPT to learn programming and successfully refactored 86% of the code smells.	No Final Prompt Provided
Wang *et al.* [10]	Chain-of-Thought, Few-Shot, Context-Specific, Output Constraints	Using Chain-of-Thought, Few-Shot, Context-Specific and Output Constraints, RETester was able to identify new bugs in codebases that was never found before.	Now, I will give the definition of the current refactoring, you need to understand it. You need to make sure the original refactoring could still be applied on the variant. 1. Refactoring Type: Definition 2. To expose more bugs in the refactoring engines, please generate edge case variant considering the Characteristic in current refactoring scenario. You need to generate the variant according to the Input Program Structure Template, it is Template. 3. You should give me the variant, the program elements to be refactored, and the procedures to refactoring. 4. The generated variant should not contain any syntax errors. The Java program you generated should conformance with the JDK Version standard. Please generate one edge case variant considering different edge usage scenarios of Characteristic based on the template. The variant format should be Format.
Midolo *et al.* [61]	Context-Specific, Output Constraints, Zero-Shot	Using Context-Specific, Output Constraints, and Zero-Shot, the authors was able to produce 90.7% accurate refactoring suggestions with GPT-4.	System: "You are a software developer, skilled in writing Python source code and refactoring Python code using [pythonic idiom]" User: "Refactor the code using the [pythonic idiom] idiom, and provide the refactored Python code along with the number of [pythonic idiom] refactoring you have made: [code]"
Ksontini *et al.* [50]	Zero-Shot, One-Shot, Few-Shot	Using Few-Shot (specifically 50-shot) the authors were able to achieve a success build rate of 63% with refactoring dockerfiles.	1. In context with Few-Shot 2. Prompt components mentioned in paper 3. Demonstration selection
Alturayeif & Hassine [42]	Context-Specific, Few-Shot, Output Constraints, Chain-of-Thought	Using Context-Specific, Few-Shot, Output Constraints and Chain-of-Thought, the authors were able to detect and refactor linguistic bad smells.	No Final Prompt Provided
Divyansh *et al.* [64]	Few-Shot	By using Few-Shot, authors was able to reduce code debt significantly.	No Final Prompt Provided
Guo *et al.* [66]	Chain-of-Thought, Context-Specific, Zero-Shot	By using Context-Specific prompting in Prompt 2, the authors were able to achieve a 83.58% similarity with human refactored code.	Prompt 1 (P1): the simplest prompt.
			Prompt 2 (P2): P1 + Scenario Description.
			Prompt 3 (P3): P1 + Detailed Requirements.
			Prompt 4 (P4): P1 + Concise Requirements.
			Prompt 5 (P5): P4 + Scenario Description.
			Examples: Prompt 1 (P1): [INST] code snippet: "'original code''' code review: review comment Please generate the revised code according to the review. In your response, put the revised code between triple backticks and avoid mentioning the programming language between the backticks. [/INST] Prompt 2 (P2): P1 + Scenario Description [INST] As a developer, imagine you've submitted a pull request and your team leader requests you to make a change to a piece of code. The original code being referred to in the hunk of code changes is: "'original code''' There is the code review for this code: review comment Please generate the revised code according to the review. In your response, put the revised code between triple backticks and avoid mentioning the programming language between the backticks. [/INST]
Dong *et al.* [65]	Context-Specific	Using Context-Specific prompting with ChatGPT, the authors was able to identify previously unknown refactoring bugs.	For Testing: Description: "Please generate ten java test programs that test the move method refactoring and meet the following constraints" Conditions: "The original class of the method to be moved contains a field of the target class type." "The name of the method to be moved is 'methodToBeMoved'." "The target class name for the method to be moved is 'TargetClass'." Features: "The method to be moved calls a private field within the original class." "The modifier of the method to be moved is 'protected'." "The target class of the method to be moved is an anonymous class." For feature extraction using ChatGPT: You are an expert in testing refactoring engines, and your task is to summarize the kind of features a test Java program should have so that it could trigger the same bug in [refactoring type] as the following test program triggers. Notably, 'features' refer to code entities with specific characters as well as relationships among such entities. The given test program that triggers a bug in [refactoring type]: [source code] The error messages of [refactoring type]: [error message] To make it clear, we present the following three examples to illustrate how to extract features that cause bugs during [refactoring type] process. Example 1 Test Program: "[source code]" Error Message: "[error message]" Summarized Features: "[features]" Example 2 Test Program: "[source code]" Error Message: "[error message]" Summarized Features: "[features]" Example 3 Test Program: "[source code]" Error Message: "[error message]"
Ghammam *et al.* [49]	Zero-Shot, One-Shot	Using One-Shot, BuildRefMiner was able to obtain a F1 score of 76% across refactoring build systems.	Task: Given the commit changes below that are applied on build system build scripts which will be delimited with delimiter characters, identify any occurrences of the listed refactoring types. Provide the results strictly in JSON format with the following keys: RefactoringType and Details. If there are multiple refactorings, return them as a list of JSON objects, where each object contains the following: - "RefactoringType": The type of refactoring detected. - "Details": A Description with further information about the change. Here are two examples of the format I expect: ... If no refactorings are detected, return the message: "No refactorings found." This is a list of 24 refactoring types in build files: ... commit

Table 5: Prompt technique used in LLM-driven refactoring (RQ_3) - Cont'd.

Study	Prompt Technique	Best Approach	Final Prompt
Batole *et al.* [56]	Chain-of-Thought	Using Chain-of-Thought, MM-Assist was able to achieve a Recall@3 of 80%, a 2.4x improvement compare to the standard state of the art tools.	No Final Prompt Provided
Palit & Sharma [11]	N/A	The paper didn't talk about best prompting methods but by using Supervised Fine-Tuning and Reinforcement Learning the Code-T5 model was able to increase unit tests passes from 41 to 66.	No Final Prompt Provided
Rajendran *et al.* [68]	N/A	The paper does not mention a best prompting approach, rather, it mentions different usages of Agent to best refactor, breaking down tasks for Specialized agents.	No Final Prompt Provided
Kim [43]	Zero-Shot	The paper identified Claude combined with Context-Specific Prompts as the most effective LLM with the highest PRQS of 89.51 followed by Meta's 88.98.	Prompt 1: Identify applicable design patterns for the given program. Prompt 2: Apply [design pattern] to
			[location] in the given program. Prompt 3: Can [design pattern] be applicable to [location]?
Wang *et al.* [44]	Few-Shot	Using Few-shot and Context-Specific prompting with GOT-2o-mini, the authors was able to reach 98% precision for refactoring identification, and 80% for refactoring.	You are a software testing expert. I will give you some historical bug reports for the refactoring engines.
			You need to extract the following information from the bug reports: 1. Refactoring type; 2. Bug symptom;
			3. Input program characteristic. The following are examples: Example. The extracted information format
			should be: Format.
Wang *et al.* [45]	Few-Shot	The paper did not talk about best prompting methods, but by using Ensemble Learning the paper was able achieve higher EM.	Multiple prompts are featured (see paper for individual prompts)
AlOmar *et al.* [69]	Few-Shot, One-Shot, Zero-Shot, Context-Specific, Output Constraints, Chain-of-Thought	By using context-specific, few-shot, and output-constrained prompting the authors was able to reduce prompting turns from 13.58 average to 1.45.	You are an expert in programming, refactoring, and providing advice in software quality. Given the following
			prompt/code snippet, modify/write the code so that it best matches what the user wants, and provide
			comments explaining your work. If needed, ask clarifying questions to check both your and the user's
			understanding of the prompt. Working Set: [input code] Context: ... Refactoring Task: ... Steps to Follow:
			1) Analyze the Code... 2) Apply Refactoring Strategies ... 3) Provide Multiple Solutions... 4) Validate
			Functionality... 5) Comment and Document... Output Format: ... Example start: [example] Example end
			If there are any unclear instructions, please ask clarifying questions.
Midolo *et al.* [46]	One-Shot	Using Context-Specific Prompting with GPT-3.5-Turbo, the authors was able to refactor 2101 out of 2132 java for-loops into stream pipelines.	System: 'You are a software developer, skilled in writing Java source code and refactoring for loops into stream-based code'
			User: 'Please refactor this code to use streams, and provide only the Java code without any markdown, explanation, or comments: *code*'
Pacho *et al.* [73]	Zero-Shot	Using Context-Specific and multi-llm agents with MetaGPT the authors was able to reduce cyclomatic complexity by 26%.	No Final Prompt Provided
Cassamitin *et al.* [70]	Chain-of-Thought, Context-Specific, Zero-Shot	By using Context-Specific and Zero-Shot prompting the authors are able to improve CodeLlama to achieve comparable performance to ChatGPT (77.13 vs 76.44 BLEU-T)	Prompt 1 (P1): the simplest prompt. Prompt 2 (P2): P1 + Scenario Description. Prompt 3 (P3): P1 + Detailed Requirements.
			Prompt 4 (P4): P1 + Concise Requirements. Prompt 5 (P5): P4 + Scenario Description.
			Examples: Prompt 1 (P1): [INST] code snippet: "'original code"' code review: review comment Please generate the revised code according to the review.
			In your response, put the revised code between triple backticks and avoid mentioning the programming language between the backticks. [/INST]
			Prompt 2 (P2): P1 + Scenario Description [INST] As a developer, imagine you've submitted a pull request and your team leader requests you to make a change to a piece of code.
			The original code being referred to in the hunk of code changes is: "'original code"'
			There is the code review for this code: review comment Please generate the revised code according to the review.
			In your response, put the revised code between triple backticks and avoid mentioning the programming language between the backticks. [/INST]
Pandini *et al.* [71]	Chain-of-Thought, Context-Specific	Using Context-Specific prompting with Claude 3.5 Sonnet the authors was able to reach up to 75% result preference from study praticipents.	Multiple prompts are featured (see paper for individual prompts)
Guo *et al.* [67]	Chain-of-Thought, Context-Specific, Few-Shot, Zero-Shot	Using Few-Shot prompting the authors was able to achieve 79% accuracy in intention extraction and 66% in code refactoring.	Multiple prompts are featured (see paper for individual prompts)
Xu *et al.* [47]	Chain-of-Thought, Context-Specific, Few-Shot, Output Constraints	Using Chain-of-Thought prompting, HLSRewriter was able to achieve an 86.67% refactoring pass rate.	Multiple prompts are featured (see paper for individual prompts)
Thackuk [48]	Zero-Shot	Using Context-Specific prompting with Claude, the authors was able to reach the highest success rate of refactoring by 78.8% which ChatGPT followed with 76.6%.	No Final Prompt Provided
Guenoune *et al.* [12]	Chain-of-Thought, Context-Specific, Few-Shot	Using Few-Shot and Chain-of-Thought with Gemini the authors was able to reach a highest score of 98.75% in matching with human naming.	No Final Prompt, but the steps are described: Step 1: Instructs the LLM and check its understanding of the RCS + dot syntax Step 2: Ask LLM its criteria about class relevance.
			Step 3: Ask LLM evaluate each concept (explanation + score) and name the candidate class Step 4: Asks the LLM to format the result
Amaral [72]	Zero-Shot	Using Zero-Shot and Context-Specific prompting, Copilot (ChatGPT-4o) was able to achieve up to 58.78% successful code smell removals.	Context:I'm refactoring test smells from a test file to improve code quality.
			Issue Details: Smell Category: Test smell category Smell Location:- Line Range: startLine: startLine, endLine: endLine or- Line: line, Column: column, Index: index
			Description: Detailed test smell description Request: Refactor the affected code to eliminate the Test smell category. Ensure the test remains correct, readable, and maintainable.
De Sousa *et al.* [74]	Chain-of-Thought, Output Constraints	Using Context-Specific prompting with GPT-4o, the authors was able to achieve a F1-scores of 79% for detecting PL/SQL bad smell.	You are an AI assistant specialized in PL/SQL code analysis and linting. Consider the PL/SQL code smells listed below:
			Unused local variable or unused parameter. Call to procedure from package DBMS_OUTPUT. Use of an IF statement instead of EXIT WHEN to exit from a loop.
			Empty code block (BEGIN NULL; END;). Parameter mode omission (parameter declaration without explicit IN, OUT, or IN OUT).
			Use of SELECT * (projection of all columns). COMMIT or ROLLBACK in a non-autonomous transaction.
			SELECT INTO statement without an associated exception handler covering potential query exceptions (e.g. NO_DATA_FOUND, TOO_MANY_ROWS).
			Analyze the provided PL/SQL code snippet and check for those code smells. If any smells are detected, present them as a list where each item matches this format:
			- Location: line(s) where the smell occurs - Issue: smell number from the list above
			If none of the specified smells are identified, respond with this sentence: "No smell detected." Here is the PL/SQL code snippet: [code]

suggestions but also the processes used to assess this accuracy in order to make sufficiently informed decisions about their reliability, usability, and ultimate adoption within the software engineering industry, research, and beyond.

Approach. To further understand the accuracy of LLM-generated refactoring suggestions in the primary studies, we report in Table 6 the following types of data analysis:

- *Quantitative:* Indicates the dataset used to assess the accuracy of LLM-generated refactoring suggestions.
- *Qualitative:* Indicates the number and type of surveyed participants.
- *Comparative:* Indicates the tools used for comparison.
- *Correctness:* Indicates the results for measuring the accuracy of LLM-generated refactoring suggestions.

We report the Correctness metrics of each study as 'Precision', 'Accuracy', 'Recall', or 'F1-measure' when explicitly mentioned in the primary studies, and as 'Unknown' if no performance metrics are provided. We mark the Qualitative column as 'No' if no qualitative analysis was done. For the Comparative column, we document the names of tools used for comparison, marking 'No' if no comparative analysis was conducted, or 'Unknown' if the tools used for comparison were not named.

Table 6: Quantitative, qualitative, and comparative analysis of LLM-driven refactoring (RQ_4).

Study	Quantitative	Qualitative	Comparative	Correctness
Gehring [13]	1 project	No	w/ ChatGPT3.5 and StarChat-β	Accuracy: 100%
White *et al.* [58]	14 prompt patterns	No	No	Unknown
Shirafuji *et al.* [15]	880 programs	No	No	Unknown
Depalma *et al.* [16]	40 files	w/ 15 students	No	Accuracy: 96.8%
AlOmar *et al.* [17]	470 commits, 69 issues, 176 files	No	No	Unknown
Chavan *et al.* [18]	447 conversations	No	No	Unknown
Pomian *et al.* [5]	1756 refactorings	w/ 18 developers	w/ JExtract	Unknown
Pomian *et al.* [6]	122 methods	w/ 16 developers	w/ JDeodorant, JExtract, REMS,	Unknown
			GEMS, SEMI, and LiveRef	
Dilhara *et al.* [7]	9325 code change patterns	No	w/ PyEvolve	F1-measure: 96.6%
Gao *et al.* [9]	6 projects	No	w/ TESTAXE, Llama3-70B,	Unknown
			and GPT-4o	
Zhang *et al.* [39]	3,311 methods	No	w/ RIdiom and Prompt-LLM	Precision: increase to 100%,
				decrease to 93.8%
				Accuracy: increase to 87.8%-100%
				Recall: increase to 87.8%-100%
				F1-measure: increase to 92.5%-100%
Choi *et al.* [57]	17 projects	No	No	Unknown
Wu *et al.* [54]	3 datasets	No	w/ PMD, JDeodorant, Organic,	Precision: decrease to 82.13%
			DesigniteJava, JSPIRIT,	Accuracy: increase to 95.14%
			FeTruth, and JMove	Recall: increase to 92.41%
				F1-measure: increase to 95.52%
Ishizue *et al.* [59]	1,800 methods	No	w/ ChatGPT	Unknown
Gautam *et al.* [75]	9 projects	No	w/ manual refactoring	Unknown
Baumgartner *et al.* [40]	1 project	No	No	Recall(Median): 48%
Cui *et al.* [51]	11169 projects	w/ 50 software engineers	w/ LLMRefactor, FeTruth, FeDeep,	Precision(AVG): increase to 93.3%
			PathMove, JDeodorant, JMove,	Recall(AVG): 85.4%
			RMove, FeGNN, and FePM	F1-measure(AVG): 80.7%
Zhang *et al.* [41]	58 projects	No	w/ PathMove, JDeodorant, JMove,	Precision: increase to 60.6%-91.4%
			and RMove	Recall: decrease to 58.8%-91%
				F1-measure: 74%-91%
Ben Mrad *et al.* [60]	116 methods	No	w/ Mistral-7B and Llama 2-7B	Unknown
Wang *et al.* [63]	11,336 projects	w/ 50 software engineers	w/ JDeodorant, SSECS,	Precision(AVG): increase to 76.8%
			and LLMRefactor	Recall(AVG): increase to 84.3%
				F1-measure(AVG): increase to 80.4%
Gao *et al.* [53]	30,000 code samples	No	w/ GPT-2, CodeGPT, CodeGEN,	Unknown
			and PolyCoder	
Cui *et al.* [52]	11,336 projects	w/ 50 software engineers	w/ JDeodorant, SSECS,	Precision(AVG): increase to 76.8%
			and LLMRefactor	Recall(AVG): increase to 84.3%
				F1-measure(AVG): increase to 80.4%
Zhang *et al.* [55]	1,204 files	No	No	Unknown
Menolli *et al.* [62]	4 projects	w/ 23 students	No	Unknown
Wang *et al.* [10]	458 bug reports	No	Unknown	Unknown
Midolo *et al.* [61]	736 projects	No	w/ SOTA benchmark	Unknown
Ksontini *et al.* [50]	600 dockerfiles	No	w/ PARFUM and manual refactoring	Unknown
Alturayeif & Hassine [42]	17 linguistic smells	w/ 2 professors	w/ manual refactoring	Precision(AVG): 57.5%
		and 11 graduate students		Recall(AVG): 100%
				F1-measure(AVG): 74%
Divyansh *et al.* [64]	1,032 files	No	No	Unknown
Liu *et al.* [38]	20 projects	w/ 3 human developers	w/ manual refactoring	Unknown
Guo *et al.* [66]	467 projects	No	w/CodeReviewer	Unknown
Dong *et al.* [65]	700 bug reports and 185 test cases	w/ original developers	w/ manual refactoring	Precision: 92.9%
Ghammam *et al.* [49]	609 projects	w/ 60 developers	w/ prompting technique	Precision: 79%
				Recall: 78%
				F1-measure: 76%
Batole *et al.* [56]	10 projects	w/30 grad students	w/JMove, FeTruth, HMove,	Recall: 80%
			and GPT-4o	
Palit & Sharma [11]	1,618 projects	No	w/Code-T5, PLBART, CodeGPT-adapt	Unknown
			and CodeGen	
Rajendran *et al.* [68]	4 agents	No	w/GPT-4, SonarQube, ESLint	Unknown
Kim [43]	2 projects	No	w/ChatGPT-4o, Claude 3.5 Sonnet, Microsoft 365 Copilot,	PIQS(AVG): 89.51%
			Gemini 1.5, Llama 3.1	
Wang *et al.* [44]	1,651 bug reports	No	w/SAFEREFACTOR and ASTGen	Unknown
			CodeT5, and Unixcoder	Precision(AVG): 96.40%
				Recall(AVG): 93.38%
				F1-measure: 94.83%
Wang *et al.* [45]	800 samples	No	w/CodeT5, CodeReviewer, T5-Review,	Unknown
			and CodeBERT	
AlOmar *et al.* [69]	470 commits, 69 issues, and 176 files	No	No	Unknown
Midolo *et al.* [46]	4 projects	No	w/previous SOTA approaches	Unknown
Pucho *et al.* [73]	2,000 files	No	w/manual refactoring	Unknown
Caumartin *et al.* [70]	467 projects	No	w/CodeLlama, Llama 2, Llama 3.1	BLEU-T: 77.13%
Pandini *et al.* [71]	5 projects	w/10 students	w/Qwen2.5-Coder, Claude 3.5 Sonnet	Unknown
Guo *et al.* [67]	15 projects	No	w/CodeReviewer and T5CR	Accuracy: 78.61%
Xu *et al.* [47]	74 files	No	w/GPT-4 Turbo	Accuracy: 86.67%
Tkackuk [48]	4 code snippets	No	No	Accuracy: 78.8%
Guenoune *et al.* [12]	2 projects	No	w/Gemini, ChatGPT, Claude,	Accuracy: 98.75%
			DeepSeek and manual refactoring	
Amaral [72]	10 projects	No	w/SNUTS.JS and Steel	Accuracy: 58.78%
De Sousa *et al.* [74]	44 code snippets	No	w/GPT-4o, Gemini 2.0, GPT-4o mini,	Precision: 70%
			and Phi-4	Recall: 93%
				F1-measure: 79%

Results. This section discusses the quantitative, qualitative, comparative, and correctness data analysis of LLM-based refactoring tools.

Quantitative. The datasets used for evaluating the accuracy of LLM-based

refactoring vary greatly in composition and size between the primary studies. As mentioned earlier in RQ_2, if the number of projects used is not explicitly reported in a study, we report the other quantities of data that are mentioned. For example, DePalma *et al.* [16] specified 40 files, each containing 15-30 lines of code. Besides full projects and files, data types include methods, commits, and even different prompt patterns, among others. The sizes of these datasets go from being as small as 1 program [13] or 116 methods [60], to as much as 11,336 projects [52, 63]. Notably, earlier studies tended to employ smaller datasets for evaluation. This is likely due to the increased capabilities of LLMs as of late, with models such as ChatGPT-4o being released on May 14, 2024. Continuous updates to these models include further token limit increases, allowing for greater dataset size. More than a third (46%) of the studies tested their tools on varying numbers of 'projects', implying the use of larger programs composed of multiple files, components, and classes, mainly from open-source repositories. However, there is still vast variability in the datasets used for evaluating LLM-based refactoring approaches. Beyond the data types mentioned above, differences in programming languages and project sources can also affect the observed accuracy of refactoring suggestions. This lack of standardization makes comparisons between studies more difficult, emphasizing the need for unified datasets in future work.

Qualitative. The reviewed primary studies also exhibited variation in their qualitative methodology. There were 13 studies (26%) that reported the use of human feedback in their evaluation [5, 6, 16, 38, 42, 49, 51, 52, 56, 62, 63, 65, 71], with the remaining 37 (74%) not mentioning any participant involvement [7, 9–13, 15, 17, 18, 39–41, 43–48, 50, 53–55, 57–61, 64, 66–70, 72–75]. Those that did employ a survey to evaluate their refactoring tool or approach had different types of participants and sample sizes. Professional developers and engineers were surveyed in eight of the studies (16%) [5, 6, 38, 49, 51, 52, 63, 65], while four (8%) studies used only student participants [16, 56, 62, 71], and one used both professors and students [42]. The number of participants throughout the studies ranged from 3-60, with the student sample sizes residing in the bottom half of the range-10, 11, 15, 23, and 30 students respectively. This variation suggests that the sample sizes may be due to the assumptions of the research teams, as well as the resources that were available to them. While the professional developers and engineers can offer knowledgeable and insightful responses due to their experience with real-world refactoring challenges, the use of student participants is also very valuable and necessary when it comes to evaluating tools or approaches designed to teach refactoring concepts. Ultimately, it is important that authors justify their choice of participants and sample size to better understand these nuanced answers and their context within the study's goals and target applications. While most studies did not employ user surveys, those that did gathered valuable insights into user experience beyond the authors' own observations, such as Cui *et al.* [51], where 68% of surveyed software engineers preferred the authors' tool HMove, over other existing alternatives. This suggests this is a beneficial dimension for future work.

Comparative. Out of 50 studies, 37 (74%) studies directly compared LLM-

generated refactorings against existing refactoring tools and baselines or manual refactoring [5–7, 9, 13, 38, 39, 41, 42, 49–54, 59–61, 63, 65, 75]. Specifically, Ghammam *et al.* [49] compared the performance of their tool using different prompting techniques. Notably, the inclusion of comparison, as well as the range and quantity of the refactoring tools used for comparison, in the primary studies can be influenced by the financial and time resources available to the researchers. The remaining 13 studies (26%) did not make comparisons [15–18, 40, 48, 55, 57, 58, 62, 64, 69], or did not specify the name of the baseline [10]. A majority of the studies did perform some comparative analysis, yet the tools and baselines used varied significantly based on individual methodologies. This lack of a standardized benchmark makes it more difficult to get a full picture of LLMs' relative performance in refactoring tasks. For instance, some used existing refactoring tools and models [5–7, 39, 41, 51, 52, 54, 61, 63], just other LLMs [13, 53, 59, 60], or both [9], and 13.46% compare performance against manual refactoring [12, 38, 42, 50, 65, 73, 75]. One such study did a comparative analysis against an existing refactoring tool and manual refactoring [50]. Wang *et al.* [45] differentiated between pretrained models (T5-Review, CodeT5, CodeReviewer, and CodeBERT) and LLMs (CodeLLaMA, StableCode3B, LLaMA3, and ChatGPT 4o) in code review tasks revealing that pretrained models preformed better than LLMs due to LLMs preforming more code changes. There were some tools that were used for comparison in multiple studies, such as JDeodorant, JMove, and PathMove, which could be a good start for establishing a standardized set of refactoring tools and baseline methods for comparison, allowing for a more consistent evaluation of LLM performance across future studies that the field would benefit from greatly.

Correctness. Each of the primary studies reported evaluation metrics for their respective tools, however, there was variation in the types of metrics that were reported. Similar to the comparative and qualitative analysis, there was a significant difference in the number of studies that reported performance metrics (e.g., Accuracy, Precision, Recall, F1-measure), 22 to be exact [7, 12, 13, 16, 39–43, 47–49, 51, 52, 54, 56, 63, 65, 66, 70, 72, 74], and those that reported quality metrics (e.g., cyclomatic complexity, lines of code, etc.) with 19 [5, 6, 9, 10, 15, 17, 18, 38, 50, 53, 55, 57–62, 64, 75]. The Accuracy and F1-score metrics reported the highest percentages out of the performance metrics and almost always increased when compared to existing refactoring tools. While one study reported an Accuracy of 58.58% [72], the vast majority of results were substantially higher with none under 78.61% and 74% respectively. Reported Recall percentages ranged from 48%-100%, with a majority being 84.3% or higher, indicating that the tools were decent at identifying valid refactoring opportunities. When compared to other automated approaches or manual refactoring, the Precision metric saw both increases [39, 41, 51, 52, 63] and decreases [39, 54], suggesting that the quality of generated refactoring suggestions varied depending on the context of each study. One interesting metric, BLEU-T, was reported by Caumartin *et al.* [70] at 77%. This measures the similarity of token sequences between the LLM generated code and ground truth code. This metric could be a useful metric to include in further studies, as it is suited

to evaluating LLM output by the basic units, tokens, by which they interpret text. Although the available performance metrics display promising results, it would be beneficial for future studies to include the more common Accuracy, Precision, Recall, and F1-measure in their evaluations so that the effectiveness of the latest refactoring tools can be compared using consistent and established benchmarks.

Figure 4 displays the count of papers that report each of the four metrics (Quantitative, Qualitative, Comparative, Correctness) from Table 6 respectively. This view more clearly expresses the distributions of methodological focus, emphasizing which types of analysis are more prominent in studies concerning LLM-based refactoring approaches. As shown in the chart, quantitative analyses can be done with all 50 papers (100%), followed by comparative evaluations at 74%, correctness-focused assessments at 44%, and qualitative analyses at 26%. This overview provides a clear snapshot of current research practices and can help guide future studies toward underrepresented evaluation approaches for more balanced and comprehensive assessments of LLM-based refactoring tools.

Summary for RQ$_4$. *Reported performance is generally high, with Accuracy and F1-scores often exceeding 78.6% and 74%, respectively, showing improvement over existing tools. While the precision of suggestions is context-dependent, the use of non-standardized benchmarks and metrics, along with a gap in user-centered evaluation (74% of studies did not integrate human feedback), makes it difficult to establish a generalizable conclusion about LLM accuracy. This indicates a need for a more unified evaluation framework that incorporates standardized benchmarks, consistent performance metrics, and diverse evaluation methods.*

4.5. RQ$_5$: What refactoring characteristics are addressed in LLM-driven refactoring?

Motivation. As LLMs become more popular in the field of refactoring, it is crucial to examine which refactoring methods LLMs excel in and which refactoring methods LLMs need to be studied more to gather data in terms of quantitative, qualitative, comparative, and correctness. Understanding this will help us best to recognize the gaps in the studies that pre-exists in the field of refactoring with LLMs. This research question is connected with RQ$_4$, as we dive deeper into the specific characteristics of refactoring we can examine them each within the quantitative, qualitative, comparative, and correctness frame.

Approach. We looked closely at each paper to examine the quantitative, qualitative, comparative, and correctness of each research study. Here is a breakdown of how we categorized each of the data points from the papers:

Quantitative. This refers to primary studies that use numerical data to assess the performance of refactoring with LLMs. For quantitative data points, if a paper has many numerical characteristic breakdowns of their studies aspects, such as the dataset size, type of data points, metrics such as Accuracy, Precision,

Recall, F1, or performance time, we qualify the study as having quantitative data points.

Qualitative. This refers to primary studies that explore developers' perceptions and motivations regarding refactoring and LLM-assisted tools, typically using non-numerical methods. For qualitative data points, if a paper mentions or focuses on human-centric evaluations, or descriptive evaluations, including developer/student feedback, user studies, expert reviews/analysis, or assessments of maintainability, readability or usability, we qualify the study as having qualitative data points.

Comparative. This refers to primary studies that systematically compare outcomes across two or more groups, typically examining LLM-assisted refactoring against human-only refactoring, traditional tools, or other LLM methods. For comparative data, if a paper involves benchmarking or performance comparisons between pre-existing baseline, such as other LLMs, human developers, or traditional refactoring tools, we qualify the study as having comparative data.

Behavior Preservation. This refers to primary studies that specifically measure the preservation of behavior and functional soundness of the generated refactoring output, ensuring that the transformation does not introduce bugs or alter the program's intended behavior. For functional validity, if a paper involves how accurately the refactoring preserves functionality, we qualify the study as having correctness data.

In collecting these points we also looked into how each study performed code smell detection, what level of refactoring they performed on, the refactoring types they performed and how did they check for behavior preservation after performing in Table 8 along with Figure 5[3]. To further explain, "Refactoring Identification" refers to the code smells that were addressed by the authors. Within "Abstraction Layer," we addressed whether the authors were talking about code level refactoring, design level refactoring or architecture level refactoring. For code level refactoring authors completed refactoring within a few lines of code inside a single file, for design level refactoring authors addressed code smells across different files, classes, and modules, and for architecture level refactoring, authors addressed refactoring on a higher level regarding folders, files, and the overall organization of the software. For "Behavior Preservation", we talked about how the authors checked for whether or not after refactoring, code stayed the same functionally, by following the categories listed in AlOmar *et al.* study [2]. Figure 5, shows a clear mapping of code smell, layer of refactoring, and what was the exact refactoring type applied with the studies; eliminating the lowest frequency data points such as Architectural Layer, Refused Bequest, Data Clumps, Long Parameter List, Lazy Class, Message Chain, Lingusitic smells, Test code Smells, Docker file code smells, Architectural code smells, Hardware code smells, and SQL code smell for a more clear and organized view.

In addition, we also pointed out the exact methods of detection and refactoring

[3]The figure is generated using Manus.

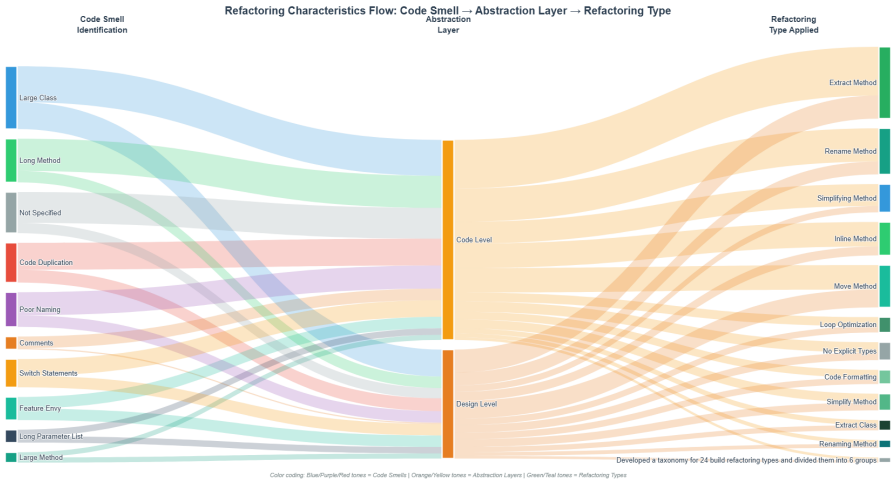

Figure 5: The relationship among refactoring type applied and code smell identification in LLM-driven refactoring studies.

to better understand what and how exactly each paper approaches refactoring in Table 9. Here are the code smells that we found, where we labeled labeled what each study found and what their definitions are in our own words, referencing Refactoring Guru[4], and also addressed special cases that do not exist on Refactoring Guru but rather defined by the papers themselves in Table 7.

Table 7: Code Smell Definitions (RQ_5).

Code Smell	Related Papers	**Definition**
Code Duplication	[15–17, 38, 48, 49, 49, 56, 64, 68, 70]	Identical code that has the same functionalities in different places of the code base.
Switch Statements	[15, 16, 55, 61, 65]	Long sequence of if statements and switch operators.
Feature Envy	[16, 41, 51, 53, 54, 56, 61]	When one class/method accesses another's fields for computation too often.
Large Class	[16, 38, 41, 49, 52–56, 62–65, 68, 73]	A class that contains too many lines of code, taking in too many fields or has too many methods.
Long Method	[5, 6, 16–18, 38, 41, 49, 57, 62, 64, 65, 68, 73]	A method that is way too long and confusing to understand.
Refused Bequest	[53, 54]	When the child class inherits the parent class's methods and properties but does not use it or is largely redefined for other purposes.
Data Clumps	[40]	Repeated code with the same groups of variables.
Poor Naming	[60, 64, 65, 65, 66, 70, 73]	When a method/class/object does not have an intuitive name.
Long Parameter List	[48, 55]	When a method holds too many parameters.
Lazy Class	[10]	When a class is too small or does not do enough to be a class of its own.
Comments	[64, 70, 73]	When a method/class has too many unclear comments.
Message Chain	[56]	When there are a long list of methods calling each other.
Lingustic smells	[42]	Specialized code smell specificlly for language; such as prompts and comments. This was not mentioned in the refactoring guru.
Test code smells	[9, 72]	Specialized code smell specifically for test cases, not mentioned in refactoring guru. Defined by the paper: Eager Test, Duplicate Assert, Assertion Roulette.
Docker file code smells	[50]	Specialized code smell specifically for Dockerfiles, not mentioned in refactoring guru. Defined by the paper: Temporary File Smells, Shell/Script Smells.
Architectural code smells	[71]	Specialized code smell specifically for code on an architectural level, not mentioned in refactoring guru. Defined by the paper: Cyclic Dependency, Unstable Dependency, God Component, Hublike Dependency
Hardware code smells	[47]	Specialized code smell specifically for hardware, not mentioned in refactoring guru. Defined by the paper: Dynamic Pointers, Recursion, and Dynamic Memory.
SQL code smells	[74]	Specialized code smell specifically for hardware, not mentioned in refactoring guru. Defined by the paper: UVP, CDO, IEW, ECB, PMO, USA, CAT, and QEH.

Results. This section discusses the quantitative, qualitative, comparative, and correctness data analysis of LLM-based refactoring tools, grouped by refactoring type: Rename Method, Extracting, Simplifying Method, Code Formatting, Split File, Inline Method, JUnit Migration, Loop Optimization, and Move Method.

Rename Method. Rename Method refactoring involves changing the names

[4]https://refactoring.guru/

of variables, functions, or classes. The studies examining this task [13, 15–17, 38, 44, 48, 50, 59, 60, 65, 66, 70, 73, 75] showed variation in quantitative dataset sizes, ranging from very small datasets, such as a single Java program in Gehring [13], to larger collections of up to 880 Python programs in Shirafuji *et al.* [15], or 600 Dockerfiles in Ksontini *et al.* [50]. While only a few studies reported correctness metrics, those that did showed generally high performance, with 100% Accuracy in Gehring [13], 96.8% in Depalma *et al.* [16], and 92.9% Precision in Dong *et al.* [65], suggesting that LLMs can reliably perform Rename Method refactoring when prompted appropriately. Five studies performed qualitative evaluations involving human developers [16, 38, 44, 48, 65, 66, 70, 73], giving feedback on generated refactoring outputs. Comparative analyses were conducted in multiple papers, either against existing refactoring tools, like ChatGPT, JDeodorant, PARFUM, or manual refactoring performed by developers [5, 13, 39, 44, 48, 59, 73, 75], often showing that LLM-based solutions can match or slightly outperform baseline approaches in identifying suitable renaming targets. Five studies [38, 50, 65, 73, 75] specifically compared LLM-based tools with manual, human refactoring, which is relevant given that human comprehension and agreement are crucial to successful renaming. LLM-based solutions sometimes matched or outperformed human experts, but human judgment was ultimately used to validate output. Overall, the breadth of studies on Rename Method refactoring demonstrates that LLM-assisted approaches are effective for renaming tasks, and their output is generally consistent with what developers would consider correct and appropriate, under guided conditions. The absence of some metrics across the papers indicates the need for structured, systematic evaluations so that stronger patterns can be identified from consolidating the studies' results.

Extracting. Extracting refactoring involves two related types: Extract Method, which moves code snippets to a new method and replaces the original code with a method call, and Extract Class, which creates a new class and places the fields and methods responsible for the relevant functionality in it [5, 6, 10, 12, 13, 15, 17, 38, 39, 44, 48, 50, 53, 54, 57, 59, 62, 64–67, 69, 70, 73, 75]. This task was the most widely studied among the refactoring types, with evaluations spanning a variety of datasets and approaches. Quantitative dataset sizes varied widely, from small sets of a single Java program in Gehring [13] to large-scale analyses of tens of thousands of code samples [53]. Correctness metrics were reported in a subset of studies, with generally high values such as 100% Accuracy in Gehring [13], 92.9% Precision in Dong *et al.* [65], and median Recall of 48% in Baumgartner *et al.* [40], suggesting that LLMs can reliably perform Extract Method refactoring when prompted appropriately. Several studies incorporated qualitative evaluations involving human developers [5, 6, 12, 38, 44, 67, 69, 70, 70, 75], which provided insight into the practical usability, maintainability, and adherence to developer expectations of LLM-generated refactorings, although most studies did not include such feedback. Comparative analyses were conducted in multiple papers, either against existing refactoring tools such as ChatGPT, JDeodorant, RIdiom, SSECS, LLMRefactor, or manual refactoring performed by developers [5, 11–13, 39, 44, 48, 52, 59, 63, 66, 67, 70, 73, 75], often show-

ing that LLM-based solutions can match or outperform baseline approaches in identifying suitable code segments for extraction. Notably, the Extract Class studies highlight the added complexity of determining which fields and methods belong together in a new class, requiring models to reason about functional cohesion and class-level organization, a task that appears more challenging than method-level extraction but still benefits from LLM guidance. The inclusion of both method-level and class-level extraction studies indicates that LLMs are capable not only of identifying extractable code segments but also of reasoning about higher-level structural organization, suggesting potential for broader software design support. Overall, the breadth of studies on Extract Method and Extract Class refactoring demonstrates the great potential of LLM-assisted refactoring in real-world contexts, though the variability in reported metrics is likely a reflection of differences in dataset sizes, composition, and evaluation methods when it comes to applying LLMs to this task.

Simplifying Method. Simplifying Method refactoring involves making method calls simpler and easier to understand, such as modifying conditionals or adjusting parameters. Evaluations of this task [10, 15–18, 47, 48, 60, 69, 73] were less extensive than for the Rename or Extract Method types. Quantitative dataset sizes varied widely, such as small sets of 40 files with student feedback in Depalma *et al.* [16], 24 tasks in Xu *et al.* [47] or large-scale analyses of hundreds of programs or conversations in Shirafuji *et al.* [15], Wang *et al.* [10], and Midolo *et al.* [61]. Notably, qualitative evaluation was more prominent in this set than in other refactoring types, with Depalma *et al.* [16] including feedback from 15 students and some studies [18, 48, 60, 69, 73] involving developers. The other metrics, correctness and comparative analysis, were pretty similar in range and results to what was observed in Rename Method and Extract Method refactoring, with most studies not reporting explicit correctness metrics and only one performing tool-to-tool comparisons. Overall, while the available studies suggest that LLMs can assist in simplifying methods, the limited number of evaluations and infrequent reporting of correctness metrics showcase a need for more large-scale studies to validate reliability and performance of LLMs when it comes to Simplifying Method refactoring.

Code Formatting. Code Formatting refactoring involves changes to code style, such as indentation or ordering functions. The small sample size of this task [16, 59, 66] meant that any solid patterns in the results of the evaluation metrics were difficult to identify. In fact, the metrics that the two studies' report are relatively inconsistent from each other. Depalma *et al.* [16] conducted a quantitative evaluation on 40 files and reported a correctness Accuracy of 96.8%. Their study also included qualitative feedback from 15 students, but did not perform a comparison against other refactoring tools. In contrast, Ishizue *et al.* [59] evaluated 1,800 programs in a large-scale quantitative analysis, with results from their three phase approach compared directly to ChatGPT, though they did not report any correctness metric nor qualitative evaluation. While both studies demonstrate that LLMs can have the capability of outputting stylistically consistent formatting, the lack of reported correctness data in [59] along with the absence of tool-to-tool comparisons in [16] make it difficult

to assess the reliability of these results. However, Guo *et al.* [66] was able to provide qualitative and quantitative data on various setting changes of ChatGPT and in comparison with the other state-of-the-art tools that exist, along with correctness checking with specific refactoring types, offering a conclusive usage of ChatGPT in refactoring. Similar to Split File refactoring, the scarcity of papers specifically discussing Code Formatting refactoring highlights the need for more widespread evaluations before coming to further conclusions about the effectiveness of LLM-based approaches for this task.

Split File. Split File refactoring involves separating functions, methods, or classes into different files for modularity. Only one study examined this type [18], mentioning that a few of the 54 refactoring-related conversations that developers had with ChatGPT had the goal of splitting the code to enhance readability and maintainability. The focus of this paper was to understand the nature of Developer-ChatGPT conversations and the model's role in the evolution of software development. No qualitative developer feedback, or correctness metric related to this specific refactoring type was reported, and comparative analysis with other refactoring tools was absent. Due to this lack of focus on Split File refactoring, further comparative analysis of LLM-based refactoring tools for the Split File task is necessary for a more concrete understanding of their true effectiveness in supporting modularity and larger-scale software refactoring.

Inline Method. Inline Method refactoring involves replacing method calls with the method body itself, removing the need for separate calls. Multiple studies have examined this task as part of their study in passing [10, 38, 44, 48, 66, 67, 70, 73, 75]. Gautam *et al.* [75] performed a quantitative evaluation with a dataset of nine Python projects, comparing LLM-produced refactorings with human developer implementations, but did not report any correctness metrics or qualitative developer feedback. Wang *et al.* [10] utilized 458 bug reports in their quantitative analysis, though no comparisons to other refactoring approaches or correctness measures were included. In contrast, Liu *et al.* [38] evaluated 20 Java projects, integrating qualitative feedback from three human developers and comparing the results of prompting different LLMs with manual refactoring by human developers. However, this study did not report explicit correctness metrics. Multiple studies [44, 48, 66, 67, 70, 73], have the authors reported quantitative, qualitative, correctness, and comparative data point on various tools refactoring. Overall, while these works collectively suggest that Inline Method refactoring with LLMs is being investigated in real-world contexts, the absence of correctness measures in some studies, makes it difficult to evaluate the reliability and effectiveness of this refactoring type.

JUnit Migration. JUnit Migration refactoring involves converting existing test code to use the JUnit testing framework. Only one study examined this task [13], evaluating a single program and comparing the results of recipe-driven refactorings with outputs from ChatGPT-3.5 and StarChat–β. The study reported an Accuracy of 100%; however, this statistic was calculated for refactorings related to incorrect switch statements (such as Java switch-case structures), not specifically for JUnit Migration. Additionally, no qualitative feedback from de-

velopers was collected. Similar to Split File refactoring, the limited empirical evidence on JUnit Migration, means that no solid patterns or conclusions can be drawn regarding the effectiveness of LLMs for this task. Thus, more extensive investigations and analyses are required before claims about the reliability and utility of LLM-assisted JUnit Migration can be made.

Move Method. Move Method refactoring involves creating a new method in the class that uses the method the most, and then moving code from the old method to the new location. The code of the original method is then replaced with a reference to the new method or removed entirely. Evaluations of this task [12, 17, 41, 44, 51, 54, 56, 62, 65–67, 69, 70] spanned a wide range of datasets, from smaller sets of 4 Java projects in Menolli *et al.* [62] to large-scale analyses of over 11,000 Java projects in Cui *et al.* [51] Metrics of correctness were reported in a subset of studies, with generally high performance where available, including Precision at 92.9% in Dong *et al.* [65], and average Precision and Recall of up to 93.3% and 85.4%, respectively, in Cui *et al.* [51]. However, in both studies, these correctness metrics were reported across the total set of refactoring cases involving multiple refactoring types, rather than being specified exclusively for the Move Method refactoring type. Several papers performed qualitative evaluations with human developers or students [12, 51, 62, 65, 67, 69, 70], with 68% of those surveyed in Cui *et al.* [51]. Finding the proposed LLM-driven tool, HMove, to be the most useful compared to two other refactoring tools. These qualitative evaluations suggest that involving human developers or students helps gauge not only the correctness but also the readability, maintainability, and practical relevance of LLM-generated Move Method refactorings. This feedback is valuable, as it indicates that LLMs could generate refactorings that are both technically accurate and aligned with developer expectations and coding standards. Comparative analyses were conducted in multiple papers, mainly against existing refactoring tools such as JDeodorant, RMove, PathMove, or manual refactoring [12, 41, 44, 51, 54, 56, 65, 67, 70], demonstrating that LLM-based approaches can match or sometimes exceed the effectiveness of traditional tools in identifying suitable methods to move. Overall, while the studies collectively indicate that LLMs show strong potential for Move Method refactoring, demonstrated by promising Precision and Recall metrics as well as supportive feedback from human evaluators, the variability in reported metrics and the limited number of qualitative assessments underscore the need for more systematic and comprehensive analyses to fully understand LLM-assisted performance in this task.

Loop Optimization. Loop Optimization refactoring involves improving the efficiency or structure of loops in code, such as minimizing iterations, unrolling loops, or replacing loops with more efficient constructs. There were limited assessments of this task, with only two studies [16, 17] examining LLM-assisted approaches for Loop Optimization. Depalma *et al.* [16] conducted a quantitative evaluation on 40 files and included qualitative feedback from 15 college students (studying computer science or related field), and reported an Accuracy of 96.8% when it came to preserving the behavior of original code segments. In contrast, AlOmar *et al.* [17] analyzed 470 commits, 69 issues, and 176 files in

a large-scale quantitative study, but no explicit correctness metrics were reported. Neither study conducted comparative analyses against other refactoring tools, limiting our ability to assess the relative performance of LLM-based Loop Optimization approaches. Overall, while these studies suggest that LLMs may have potential for automating Loop Optimization, the small number of these studies suggests that before stronger conclusions about LLM effectiveness in this refactoring type can be drawn, more thorough and structured investigations are needed.

Table 8: Refactoring characteristics in LLM-driven refactoring (RQ_5).

Study	Refactoring Identification	Abstraction Layer	Refactoring Types	Behavior Preservation
Gehring [13]	Not Specified	Design Level	Rename Method, Junit Migration	Manual Analysis
White *et al.* [58]	Not Specified	Design Level	No Explicit Refactoring Types	Manual Analysis
Shirafuji *et al.* [15]	Code Duplication, Switch Statements	Code Level	Simplifying Method, Rename Method, Extract Method	Manual Analysis
Depalma *et al.* [16]	Code Duplication, Feature Envy, Large Class, Long Method, Switch Statements	Code Level, Design Level	Rename Method, Code Formatting, Loop Optimization, Simplifying Method	Static Code Analysis
AlOmar *et al.* [17]	Long Method, Code Duplication	Code Level	Rename Method, Extract Method, Move Method, Loop Optimization, Simplifying Method	Manual Analysis
Chavan *et al.* [18]	Long Method	Code Level	Split file, Simplifying Method	Manual Analysis
Pomian *et al.* [5]	Long Method	Code Level	Extract Method	Static Code Analysis
Pomian *et al.* [6]	Long Method	Code Level	Extract Method	Static Code Analysis
Dilhara *et al.* [7]	Not Specified	Code Level	No Explicit Refactoring Types	Static and Dynamic Code Analysis
Gao *et al.* [9]	Eager Test, Duplicate Assert, Assertion Roulette	Code Level	No Explicit Refactoring Types	Manual Analysis
Zhang *et al.* [39]	Not Specified	Code Level	Extract Method, Simplify Method	Manual Analysis
Choi *et al.* [57]	Long Method	Code Level	No Explicit Refactoring Types	Manual Analysis
Wu *et al.* [54]	Refused Bequest, Large Class, and Feature Envy	Code Level, Design Level	Extract Method, Move Method	Manual Analysis, Static Code Analysis Tool
Ishizue *et al.* [59]	Not Specified	Code Level	Rename Method, Extract Method, Simplifying Method, Code Formatting	Manual Analysis
Gautam *et al.* [75]	Not Specified	Code Level	Extract Method, Inline Method, Rename Method	Manual Analysis, Static Code Analysis
Baumgartner *et al.* [40]	Data Clumps	Code Level, Design Level	Extract Class	Manual Analysis
Cui *et al.* [51]	Feature Envy	Code Level, Design Level	Move Method	Manual Analysis
Zhang *et al.* [41]	Feature Envy, Long Method, Large Class	Code Level, Design Level	Move Method	Static Code Analysis
Ben Mrad *et al.* [60]	Poor Naming	Code Level	Rename Method	Manual Analysis
Wang *et al.* [63]	Large Class	Code Level, Design Level	Extract Class	Manual Analysis, Static Code Analysis
Gao *et al.* [53]	Large Class, Feature Envy, Refused Bequest	Code Level, Design Level	Extract Method, Simplify Method	Manual Analysis, Static Code Analysis
Cui *et al.* [52]	Large Class	Code Level, Design Level	Extract Class	Manual Analysis, Static Code Analysis
Zhang *et al.* [55]	Long Parameter List, Long Method, Large Class, Switch Statements	Code Level, Design Level	No Explicit Refactoring Types	Manual Analysis, Static Code Analysis
Menolli *et al.* [62]	Duplicate code, Long Method, Large Class	Code Level, Design Level	Move Method, Extract Class, Extract Method	Manual Analysis, Static Code Analysis
Wang *et al.* [10]	Lazy Class	Code Level, Design Level	Extract Method, Inline Method, Simplifying Method	Manual Analysis, Static Code Analysis
Midolo *et al.* [61]	Switch Statements, Feature Envy	Code Level	Simplifying Method	Manual Analysis
Ksontini *et al.* [50]	Temporary File Smells, Shell/Script Smells	Design Level	Extract Method, Rename Method	Manual Analysis, Dynamic Code Analysis
Alturayeif & Hassine [42]	Linguistic Smells	Design Level	No Explicit Refactoring Types	Manual Analysis
Divyansh *et al.* [64]	Code Duplication, Long Methods, Poor Naming, Comments, Large Class	Code Level, Design Level	Extract Method	Manual Analysis, Static Code Analysis
Liu *et al.* [38]	Code Duplication, Large Class, Long Method	Code Level	Extract Method, Rename Method, Inline Method	Manual Analysis, Dynamic and Static Code Analysis
Guo *et al.* [66]	Poor naming	Code Level	Rename Method, Move Method, Extract Method, Code Formatting, Inline Method	Manual Analysis, Static Code Analysis
Dong *et al.* [65]	Poor Naming, Switch Statements, Large Method, Large Class	Code Level, Design Level	Rename Method, Move Method, Extract Method, Inline Method	Manual Analysis, Dynamic and Static Code Analysis
Ghammam *et al.* [49]	Code Duplication, Large Class, Long Methods	Code Level, Design Level	Developed a taxonomy for 24 build refactoring types and divided them into 6 groups	Manual Analysis
Batole *et al.* [56]	Feature Envy, God Class, Duplicated Code, Message Chain	Class Design	Move Method	Static Code Analysis
Palit & Sharma [11]	Not Specified	Code Level	Extract Method	Manual Analysis, Dynamic and Static Code Analysis
Rajendran *et al.* [68]	Long Methods, Large Classes, Duplicated Code	Code Level, Design Level, Architecture Level	No Explicit Refactoring Types	Manual Analysis, Static and Dynamic Code Analysis
Kim [43]	Not Specified	Design Level	No Explicit Refactoring Types	Dynamic Code Analysis
Wang *et al.* [44]	Not Specified	Code Level	Inline Method, Move Method, Extract Method, Move Method, Rename Method	Manual Analysis, Static and Dynamic Code Analysis
Wang *et al.* [45]	Not Specified	Code Level	No Explicit Refactoring Types	Manual Analysis
AlOmar *et al.* [69]	Not Specified	Code Level, Design Level	Extract Method, Move Method, Simplify Method	Manual Analysis
Midolo *et al.* [46]	Not Specified	Code Level	No Explicit Refactoring Types	Manual Analysis, Static Code Analysis
Pucho *et al.* [73]	Large Class, Long Method, Poor Naming Comments	Code Level	Rename Method, Extract Method, Inline Method, Simplifying Method	Static Code Analysis
Caumartin *et al.* [70]	Poor Naming, Duplicated Code, Comments	Code Level	Rename Method, Extract Method, Inline Method, Move Method	Static Code Analysis
Pandini *et al.* [71]	Architectural Smells: Cyclic Dependency, Unstable Dependency, God Component, Hublike Dependency	Architecture Level	No Explicit Refactoring Types	Manual Analysis
Guo *et al.* [67]	Not Specified	Code Level	Extract Method, Inline Method, Move Method	Manual Analysis
Xu *et al.* [47]	Hardware level refactoring: dynamic pointers, recursion, and dynamic memory	Code Level, Design Level	Simplifying Method	Static Code Analysis
Tkachuk [48]	Long Parameter List, Poor Naming, Code Duplication	Code Level, Design Level	Simplify Method, Renaming Method, Extract Method, Inline Method	Manual analysis
Guenoune *et al.* [12]	Not Specified	Design Level	Extract Method, Move Method	Manual Analysis
Amaral [72]	Specific test code smells identified by the paper: Conditional Test Logic, Overcommented Test, Suboptimal Assert, Test Without Description, Sensitive Equality, Assertion Roulette, Duplicate Assert, Magic Number, Lazy Test Redundant Print	Code Level	No Explicit Refactoring Types	Manual Analysis, Static Code Analysis
De Sousa *et al.* [74]	Specific SQL code smells identified by the paper: UVP, CDO, IEW, ECB, PMO, USA, CAT, QEH	Code Level	No Explicit Refactoring Types	N/A

Summary for RQ_5. *By addressing refactoring characteristics in LLM-driven refactoring in the frame of the various data points, we found that out of all of the code smells, Large Class and Long Method is the most often spoken code smells and that Extract Method is the most often seen Refactoring type applied. Extracting refactoring method is the most widely seen and studied refactoring method with high Accuracy rates reported when using LLMs to refactor showing the most promising results.*

4.6. RQ_6: What are the key challenges in automating refactoring using LLMs?

Motivation. As the use of LLMs in software engineering expands, their potential to assist in automating refactoring has become a point of focus in research. Refactoring is a necessary process for maintenance and improving

Table 9: Study Methods of Correction and Detection (RQ$_5$).

Study	Method of Detection	Method of Correction
Gehring [13]	Pattern-based static code analysis using OpenRewrite recipes	OpenRewrite recipes using lossless semantic tree using visitor pattern
White *et al.* [58]	Using ChatGPT and structured prompt patterns to detect code smells	Using ChatGPT and structured prompt patterns to refactor
Shirafuji *et al.* [15]	Using radon for CC calculation, calculated Levenshtein, counted LOC, Chars, Tokens with tokenize, checked if the code compiled, and validated against hidden AOJ test cases	Collected correct programs as examples from Aizu, performed one-shot and few-shot prompting with examples
Depalma *et al.* [16]	N/A	Prompt Engineering, with focuses on prompting structures and grammar
AlOmar *et al.* [17]	N/A	The authors examined DevGPT interactions; Prompt Engineering approaches such as structures and grammar
Chavan *et al.* [18]	N/A	The authors examined DevGPT interactions; Prompt Engineering approaches such as structures and grammar
Pomian *et al.* [5]	N/A	EM-Assist builds a few-shot prompt with the selected method and send it to the LLM several times to gather possible refactoring ways, using IntelliJ IDEA to check for safety, executes one of the options
Pomian *et al.* [6]	N/A	EM-Assist builds a few-shot prompt with the selected method and send it to the LLM several times to gather possible refactoring ways, using IntelliJ IDEA to check for safety, executes one of the options
Dilhara *et al.* [7]	N/A	Mined for CPATs with R-CPATMiner, generate variants via LLMs, PyCraft validated the variants via static and dynamic checks, using PyEvolve turns valiants and original code to generalized rules, using these PyCraft automatically rewrote matching code fragments
Gao *et al.* [9]	N/A	Build an test refactoring knowledge base, UTRefactor then used the code base to Chain-of-Thought prompting to refactor
Zhang *et al.* [39]	Define smells via ASTscenario, ASTcomponent and conditions	Generate prompts using Prompt-LLM depending on the smells detected, reviewed by humans, then invoke ARIs to rewrite code
Choi *et al.* [57]	Using Lizard library to measure CC, the highest CC is treated as refactoring target	Prompting LLM for refactoring, check with human-tests and generated-tests, check for compilability, recalmates CC, repeat 20 times
Wu *et al.* [54]	Using Mixture of Experts (MoE) such as PMD and JDeodorant to find the code smell	iSMELL uses LLMs for refactoring from the detection results by creating structured prompts
Ishizue *et al.* [59]	N/A	Refractory uses ChatGPT to improve automatic program repair via analyzing structure and syntax
Gautam *et al.* [75]	N/A	RefactorBench sets up LLMs, references, and AST testing with LM agents such as SWE-agent to refactor the code base
Baumgartner *et al.* [40]	Using AST, deterministic rules, and LLMs to detect data clumps/code smells	After clumps are detected, LLM performs refactoring on the data clumps, and validates via CI/CD
Cui *et al.* [51]	By creating an inter-class code entity dependency hypergraph using pre-train code models and LLMs, where the nodes are the methods and fields and the edges are dependencies, the paper detects feature envy specifically	The trained models output a set of Move Method candidates, the ChatGPT-3.5 completes code analysis and applied via building a VS code prototype
Zhang *et al.* [41]	N/A	MoveRec uses static analysis, LLMs, and semantic similarity to decide how to use move method refactoring to directly target feature envy
Ben Mrad *et al.* [60]	N/A	Using LLMs to suggest new method names after code modification by referencing a dataset of 116 real world methods
Wang *et al.* [63]	HECS detetcts extract class opportunities by building intra-class dependency hypergraphs using pre-trained code models, and training an enhanced version of the hypergraphs to identify clusters of methods/fields	N/A
Gao *et al.* [53]	N/A	RRG framework refactors by finetuning a CodeT5-style model to compressing relevant code snippets, then using PPO-based reinforcement learning to align with preferences to generate refactored code
Cui *et al.* [52]	HECS detetcts extract class opportunities by building intra-class dependency hypergraphs using pre-trained code models, and training an enhanced version of the hypergraphs to identify clusters of methods/fields	N/A
Zhang *et al.* [55]	Pysmell automatically detected code smell via the Turning Machine Strategy	Created three types of prompts to refactor code by using Copilot, then reevaulated by Pysmell again
Menolli *et al.* [62]	Manual and using tools such as JDeodorant and SonarLint to identify code smells	Students used ChatGPT to refactor their code and used it to aid learning programming
Wang *et al.* [10]	N/A	RETESTER uses historical bug reports help train and guide LLMs using few-shot prompting
Midolo *et al.* [61]	N/A	Using ChatGPT-4 with prompt engineering (system and user) to refactor non-idiomatic python code to idiomatic python code
Ksontini *et al.* [50]	N/A	With refactored dockerfiles examples collected, they used ChatGPT-4o with few-shot prompting to generate refactored dockerfiles
Alturayeif & Hassine [42]	By using LLMs and NLPs with prompt engineering the authors was able to to identify linguistic based smells	GPT was used with custome prompts (system, user) for each smell to be refactored
Divyansh *et al.* [64]	Used SonarQube to automatically detect code smells	The code was automatically refactored via a pipeline with scripts and datasets to refactor by using the ChatGPT API
Liu *et al.* [38]	With LLMs and prompt engineering the authors asked for refactoring opportunities	With LLMs and prompt engineering java files are automatically rewritten according to specific refactoring tasks
Guo *et al.* [66]	Using existing tools the authors randomly selected 400 samples and manually annotated them for refactoring opportunities	Author utilized ChatGPT and prompt engineering by using the code and the review messages with Zero-Shot prompting
Dong *et al.* [65]	N/A	By using Chat-GPT and prompt engineering to generate test programs and analyzing the bug reports, the authors created a feature library, then used GPT to produce new test programs from the results of feature library
Ghammam *et al.* [49]	BuildRefMiner used 725 build-related commits and GPT-4o with prompt engineering to detect and classify refactorings that happened in version controls	N/A
Batole *et al.* [56]	MM-Assist uses LLMs with Chain-of-Thought reasoning to identify code smells	MM-Assist filters out valid methods then sends the most problematic methods using vectors then using LLMs to rank which methods to move via Chain-of-Thought
Palit & Sharma [11]	The authors used RefactoringMiner and SEART to detect code smells to create their dataset	The authors generated refactoring by fine-tuning code-t5 and plbart models based off of proximal policy optimization reinforcement learning training on good and bad behaviors
Rajendran *et al.* [68]	The authors mentions of using a maintainability agent who specicializes in static analysis, rule-based smell detection and LLM fine tuning specifically on maintainability patterns	The authors mentioned the use of multiple specialized LLM agents that can do different parts of refactoring such as agents who are specialized in their specific parts, they each suggest refactoring and collaborate and negotiate
Kim *et al.* [43]	The authors used formal predicate logic patterns to detect code smells	The authors used prompt engineering to refactor based off of the patterns in designs that was detected
Wang *et al.* [44]	The authors manually analyzed 518 real bug reports	The authors refactored using LLMs and prompt template and manually tagging and labeling bugs, with ten-shot prompting
Wang *et al.* [45]	The authors used a Unidiff-based quantitative metric, that compared target and real code changes to detect code smells	The authors used fine-tuning or few-shot methods with three open-source LLMs: CodeLLaMA, LLaMA3, and StableCode3B
AlOmar *et al.* [69]	N/A	The authors manually examined conversations between developers and ChatGPT and analyzed how refactoring was carried out by ChatGPT
Midolo *et al.* [46]	The authors used Abstract Syntax Tree analysis with JavaParser to automatically extract loops to be refactored	Using ChatGPT-3.5 Turbo and prompt engineering the authors was able to refactor the code
Pucho *et al.* [73]	The authors feed the code file to the Code Quality Agent which performs static code analysis and reports code smells	The Refactoring Strategy Agent takes in the report from the Code Quality Agent and planned refactoring editing and sent it to the Code Transformation Agent to apply it to the code
Cammartin *et al.* [70]	The authors collected samples from the CodeReview with code smells identified in code reviews	The author adjusted temperature settings and tested five different prompt templates with the LLM models Llama 2 and CodeLlama to refactor the code
Pandini *et al.* [71]	The authors used Arcan, that automatically detected Architectural Smells. Arcan uses denedency graph analysis and machine-learning based assessment	The authors experimented with prompts (detailed vs not), using retrieval-augmented generation for context, and different LLMS such as Qwen and Claude Sonnet
Guo *et al.* [67]	The authors used code review comments along with prompt engineering to extract code smells	Utilitizing prompt engineering and code reviews, the authors tried different methods of prompting with retrieval-augmented generation
Xu *et al.* [47]	C/C++ code is compiled using the Hardware Level Synthesis tool and errors are flagged	Using LLMs with Retrieval-Augmented Generation the authors were able to create prompts that helped them create loops instead of recrusions, pointers to arrays, static allocations instead of malloc() etc
Thachuk [48]	N/A	The authors compare Gemini, ChatGPT and Claude to compare how each did in refactoring, the authors also asked for reasons why they refactored the code
Gnenoume *et al.* [12]	The authors find where is needed to refactor by using Formal Concept Analysis and Relational Concept Analysis, that together they can find redundant or overlapping code	THE RCA and FCA builds on each other iteratively and using LLMs the authors was able to refactor the code
Amaral [72]	The authors used Steel and SNUTS.JS to detect and find code smells (rule-based static analysis)	The authors opened the code in VSCode and manually prompted Copilot and CodeWhisperer with zero-shot prompting to refactor the code
De Sousa *et al.* [74]	The authors used GPT-4o mini and Phi-4 compared to GPT-4o and Gemini 2.0 Flash by using set prompt templates with the identified SQL code smells; the best ones were the GPT-4o and Phi-4 with F1 score of 79%.	N/A

code quality, but it requires a deep understanding of context, intent, and overall codebase structure. Researchers have begun evaluating the effectiveness of LLM-based refactoring tools and approaches by measuring key metrics such as lines of code (LOC), cyclomatic complexity (CC), and Precision and Recall rates among others. While LLMs show promise in generating refactoring suggestions, their reliability and effectiveness are hindered by several common key challenges. Understanding these challenges is essential to determine what aspects of LLM-based refactoring can be improved to advance the automation of refactoring within software engineering.

Approach. To examine this, we consolidated the results of all primary studies to identify the most prevalent challenges that researchers encountered when leveraging LLMs for refactoring tasks. To prevent bias, two authors independently analyzed the primary studies to identify and extract challenges. Each researcher performed an initial, independent categorization of these challenges. The authors then convened to reconcile their categorizations through

a structured discussion, resolving differences by consensus to establish a final, agreed-upon taxonomy. For example, the author initially disagreed on whether to include Pandini *et al.* [71] in the category concerning LLMs struggling with complex refactoring tasks, but after discussion, they unanimously agreed to include it.

Results. This section discusses the key challenges encountered when automating refactoring using LLMs.

LLMs generate erroneous refactored code and are unreliable. Despite their impressive natural language processing abilities, LLMs can often produce refactored code with errors, making them unreliable for automated refactoring if not properly validated. This behavior is displayed in many of the primary studies, such as [5], which explored the effectiveness of an LLM-based Extract Method refactoring tool and found that 76.3% of LLM suggestions were hallucinations–57.4% were syntactically incorrect while 18.9% were illogical (e.g., suggesting to extract an entire method body). Similarly, when tasked with creating Python files, GitHub Copilot produced code smells in about 15% of its generated code [55]. More concerning, one study evaluated ChatGPT's ability to refactor for-loops, finding that only 28.8% of its refactored outputs were compilable noting ChatGPT's difficulty handling complex control flows and implicit dependencies [46]. These findings call into question its usefulness for refactoring production-level code.

Additionally, ChatGPT was shown to struggle with context awareness, leading to it misinterpreting the developer's original intent and, in some cases producing bugs [17]. For instance, DePalma *et al.* [16] found that ChatGPT frequently struggles to identify quality attributes like coupling and complexity and is inconsistent with performance as its effectiveness varies depending on the complexity of the refactoring task. Furthermore, Liu *et al.* [38] observed that Gemini and ChatGPT produced buggy refactored code 5% and 7.2% of the time, respectively, when tasked with refactoring Java files. Although the refactoring tool iSMELL [54] did increase the detection of code smells, LLMs sometimes fail to detect complex code smells such as feature envy, even though they excel in handling simple refactoring tasks. This inconsistency is reflected in De Sousa *et al.* [74], where F1-scores for detecting unused variable or parameter smells dropped as low as 33% (GPT-4o mini), perhaps due to ambiguity in the model's understanding of the code smell, and in Batole *et al.* [56], where up to 80% of LLM recommendations for Move Method refactoring were hallucinations exhibiting significant difficulty in suggesting appropriate target classes and demonstrating serious unreliability. Kim *et al.* [43] reveals that LLMs excel at implementing simple design patterns like Strategy but results are less consistent with more complex patterns like Factory Method and Observer. More concerningly, when performing Code Refinement Before Review (CRB) tasks, which involve evaluating and refining code changes prior to manual review, LLMs exhibited an extremely low Real Change Accuracy (RCA), averaging only 17.5% and 12.5% across datasets, indicating that most modifications introduced were unnecessary [45]. Broadly, while LLMs can be effective tools for refactoring, their ability to produce reliable code is limited by their tendency to hallucinate,

causing unintended modifications or errors.

LLMs struggle with complex refactoring tasks involving multiple files or larger codebases. While LLMs can effectively refactor code at a smaller scale, they struggle with more complex refactoring tasks that involve multiple files or large codebases, leading to incomplete or incorrect modifications if not carefully guided. This is evident in several of the primary studies, which found that LLM performance varied significantly based on task complexity [15]. Similarly, it was observed that LLMs had difficulty handling larger codebases, failing to apply changes when working with multiple files [7]. In Batole *et al.* [56], it is highlighted that Move Method refactoring is particularly difficult to implement because it requires project-level reasoning, but currently LLM's prompt and input size limitation hinder global project reasoning. Furthermore, increasing the input size and passing the complete project as input distracts the LLM by introducing less relevant information requiring additional pre-processing in the form of semantic-based analysis and Retrieval Augmented Generation (RAG) before applying LLMs for Move Method refactoring eliminating hallucinations and increasing precision.

Structured prompting and Chain-of-Thought approaches have been shown to improve LLM performance compared to general approaches, indicating the need for continuous context reminders for more complex multi-file refactoring. For example, UTRefactor, which employs a structured, Chain-of-Thought prompt technique, achieved a significantly higher test smell elimination rate (89-91%) compared to general LLM approaches (55-57%) [9]. White *et al.* [58] emphasized that while prompt techniques such as Few-Shot can aid LLMs' ability to operate on code, they can still produce inaccurate code examples if they encounter scenarios that were not represented in the given examples, which is likely when working with larger codebases. Choi *et al.* [57] demonstrated that using LLMs for iterative refactoring helped decrease the overall cyclomatic complexity of the tested projects. However, 45.5% of the generated refactored methods were initially non-plausible, highlighting the limitations of LLMs in producing reliable refactoring suggestions without additional validation.

Similarly, Gehring [13] found that both StarChat-β and ChatGPT produced incorrect refactoring instances when evaluated against the OpenRewrite tool. Additionally, these models struggled to output complete refactored code due to token limits, indicating that they are not ideal for refactoring tasks involving extremely large codebases. In addition, the effectiveness of LLMs in refactoring greatly depends on the prompting technique that is used. In one study, a One-Shot prompt proved to be the most effective approach for refactoring Java code, achieving a 34.51% unit test pass rate and a 42.97% code smell reduction rate, outperforming the 32.22% pass and 42.34% reduction rates for Chain-of-Thought and the 28.36% pass and 39.45% reduction rates of Zero-Shot [8]. Ultimately, the current inability of LLMs to grasp and retain the context of larger, multi-file codebases limits their effectiveness in refactoring. However, applying prompt techniques such as One-Shot and Chain-of-Thought can help reduce these limitations, though their reliability must be carefully reviewed.

LLMs misunderstand developers' true intent and fail to detect quali-

fied refactoring instances. Another key challenge in LLM-driven refactoring is their tendency to misunderstand a developer's true intent, leading to incorrect identification of refactoring opportunities or failure to detect qualified instances. As previously mentioned, one study found that LLMs often lack the necessary context awareness to make informed refactoring decisions [17]. Their misunderstanding is further evidenced by AlOmar *et al.* [69], where unstructured developer prompts required an average of 13.58 conversational turns to reach a satisfactory result, indicating a lack of understanding of the developer's intentions in initial prompts. This combined with their limited refactoring abilities when working with larger codebases means that they struggle with grasping complex instructions, such as identifying possible refactoring instances. For example, ChatGPT was found to frequently fail in detecting code smells and therefore missed qualified refactoring opportunities, rendering it an unreliable educational tool on its own [62]. Similarly, Tkachuk *et al.* [48] observed that general-purpose models struggled to identify problematic code, with both Gemini 2.0 Flash and Claude 3.7 Sonnet achieving only a 50% success rate in detecting "bad" code or code smells. Additionally, LLMs were observed to focus on individual methods rather than considering the broader project structure, [6]. This narrowed-down focus can restrict their ability to identify refactoring opportunities across multiple files. Cui *et al.* [51] highlighted the importance of using short, concise prompts, timely feedback, and Few-Shot prompting to improve the accuracy of LLMs in detecting refactoring candidates, showing that LLMs still require very careful guidance to avoid misinterpretation. Zhang *et al.* [39] found that when their approach failed, it was due to LLMs either producing poor results when refactoring was too complex, or they failed to detect valid refactoring opportunities because they misinterpreted the provided code.

Similarly, Alturayeif and Hassine [42] also noted that in a few cases, LLMs misunderstood their task of refactoring linguistic elements such as complex sentences, opting to explain a concept that was mentioned in the prompt instead. Chavan *et al.* [18] analyzed conversations between developers and ChatGPT and found that the model often makes its own assumptions about the output format and the quality attributes to enhance, even when not explicitly instructed to do so, which might not match the developer's original intent. Overall, these findings indicate that developers should make use of very structured and detailed prompts when interacting with LLMs, especially when it comes to recognizing refactoring opportunities.

Observations. The analysis reveals the percentages of challenges across the primary studies. The most prevalent issue, observed in 25 out of 50 papers (50%), was LLMs generating erroneous or unreliable code, particularly in studies focused on transformations of many code components. This unreliable output often manifests as hallucinations, where suggestions are syntactically incorrect or illogical. For example, Midolo et al. [46] used ChatGPT to refactor Java for-loops; however, only 29% of the automatically refactored loops compiled successfully, while the remaining 71% failed due to contextual or syntactic issues. Struggles with complex refactoring tasks involving multiple files were reported in 22 out of 50 studies (44%), a challenge that was especially preval-

ent for studies where refactoring required more global project context. This is evident in the refactoring of Cyclic Dependencies (CDs), where performance declined as complexity increased (e.g., CDs of size 3 or 4) [71]. Meanwhile, LLMs misunderstanding developers' true intent and failing to detect qualified refactoring instances impacting 17 out of the 50 papers (34%), a shortcoming most common in evaluations where refinement tasks came with vague instructions. In one study on automated code refinement, the main root causes for ChatGPT's underperformance were identified as a lack of domain knowledge, unclear location, and ambiguous changes in the review comments [66]. Overall, the prevalence of these challenges highlights that the reliability of LLMs for refactoring is highly dependent on the complexity of the task and the specificity of the guidance provided.

Summary for RQ$_6$. *LLMs sometimes produce erroneous code, struggle with complex refactoring tasks and larger codebases, and misunderstand developers' intent. To mitigate these limitations, developers must be very clear and intentional when crafting their prompts, check refactored code for errors specifically for more complex refactoring tasks, and implement techniques to decrease the size of codebases provided to LLMs.*

5. Discussion and Open Issues

Lack of Standardization and Definitions on Refactoring Method Types. One of the key challenges in the field of refactoring research, we noticed in RQ$_1$, is the lack of standardized terminology and definitions of refactoring methods. Across different studies and refactoring tools, the same refactoring techniques may be referenced under different names, described without being identified, or grouped in different ways based on individual interpretations. For example, instead of describing code changes by "reduce the software complexity" [15] researchers should have directly mentioned the usage of "Simplifying Method" or "Extract Method", or instead of mentioning the bad smell of "god class," or "god class refactoring" [54] researchers should have mentioned "Extract Class" to address god classes. This inconsistency creates difficulties in comparing and reviewing research findings, as well as evaluating the impact of specific refactoring techniques on software quality. This is especially true with the usage of LLM-driven refactoring, as LLMs trained on different datasets may struggle to correctly interpret refactoring requests due to inconsistency in naming conventions across sources. LLMs need standardized and more concrete definitions for specifications and efficiencies.

What are the Best Method Combinations for Prompt Techniques. From our observations in RQ$_3$, we know that the current existing prompt techniques are Zero-Shot, One-Shot, Few-Shot, Chain-of-Thought, Context-Specific, Output Constraints, and Ranking. We know about the efficiency of each of these

prompting techniques, and we know the efficiency of their combinations, such as Few-Shot and Context-Specific [6], Few-Shot and Chain-of-Thought [5], or Chain-of-Thought and Output Constraints [58]. So, the question lies in what is the best combination of prompting techniques to deploy to ensure the best output? We know from studies that Zero-Shot has a unit test pass rate of 28. 36%, One-Shot has 34. 51%, and Chain-of-Thought has 32. 22% [8]. In another study, Shirafuji *et al.* discussed Few-Shots, showing that between Zero-Shot, One-Shot, and Few-Shots, Few-Shot performs the best [15], and in a study about built system refactoring, Ghammam *et al.* found that One-Shot outperformed Zero-Shot [49]. This calls for extensive research to be done to more effectively decide which prompting techniques in combination will yield the best results. In these separate studies, due to the differences in datasets (how they are collected and the quality of the files) and how success is determined (differences in unit tests in testing for quality or programmers determining what is "clean" code) it is challenging to make a complete comparative empirical analysis to determine the best combination. The most effective way to successfully come to a conclusion on this would require expensive research done with a large dataset and the same tests and checks, but different technique combinations with an iterative study.

Future Considerations for Approaches for LLM-driven Refactoring. It has been shown in RQ_1 that when interacting with an LLM for refactoring there are various number of LLMs to choose from. Then, it also matters which LLM is used, for example, GPT-4 Turbo, GPT-3.5 Turbo, and Gemini, which have all shown different results in code refactoring in terms of efficiency and correctness [75]. This could mean various things, such as developers having different needs for each LLM. Are certain LLMs better at specific refactoring types than the other refactoring types? These topics have not been sufficiently explored yet.

Effects of Developing LLMs on Refactoring Usage. Another key point to consider when developing an efficient way to work with LLMs is the LLM's own development. In RQ_6, we asked about the key challenges in using LLMs, we can also consider it possible to be solved or worsened by developing LLMs. As an example, just between two versions of ChatGPT, such a ChatGPT-4 and ChatGPT-3.5, a decline in performance in various topics, such as math, answering sensitive questions, and prompt instabilities, has been observed. Within our topic, it has been found that from 3.5 to 4, ChatGPT has gotten worse at adhering to formatting instructions [86]. This is a critical problem with using LLMs for refactoring, as failure to adhere to formatting can lead to compilation issues.

Possible Language and Dataset Biases in LLMs. As LLMs become more internationally well-spread, different LLMs will be trained on different datasets, especially datasets that contain other languages. As we know, LLMs such as ChatGPT are capable of conversing in other languages, but it has been reported that replies fluctuate in quality across different languages. ChatGPT is known to hold certain biases in conversation, even in English, biases within

gender, politics, and race[5] are present. While those social problems held by ChatGPT are out of the scope of our paper, it is important to note if LLMs might also have biases in writing code and refactoring code due to their training sets. In RQ_2, we examined the datasets and benchmarks used by different research papers, and one of the important analyses that have been brought forth by Shirafuji *et al.* [15] is the fact that code comments that are not written in English have noticeably been reduced, while English comments have been increased (95.11% to 97.16%). This change in proportion is due to the incorrect deletion of comments and translation of comments, both of which could lead to unnecessary and incorrect code refactoring.

Further Studies on Less Well-Known Techniques and Possible Tools. While there are existing prompting techniques that are promising, it is important to explore further for new and more efficient techniques in an ever-growing and evolving LLM field. One prompting technique that we have noticed during RQ_3 that is less well-known and should be explored more is the Ranking technique. Where the developer would ask the LLM to provide multiple different options, or refactoring code along with a ranking of the code refactoring done [5, 6]. In addition to techniques, one other tool that can be seen as useful in the world of refactoring is the application of agentic AI. As agentic tools being further refined in recent years along with the creation of memory tools and other tools made available via Model Context Protocols such as search functions, access to cloud tools, access to databases, and more, refactoring can be taken to the next step, where LLMs can be made to being capable of self-fixing, compiling, checking, and running code. Enabling multi-step thinking and processing with tool access can help the LLM from making less mistakes and avoid hallucinations. Studies have been done on possibilities of conducting multi-agent systems, where there can be multiple LLM based agents can be communicating, checking and comparing work with each other based off of each of their own responses for software engineering related work. Including taking on roles in the software development life-cycle such as requirements engineering, code generation, Quality Assurance (QA) and maintenance. Agents can simulate different types of software models as well, such as waterfall, and agile methods [24]. We hope to see that this can be studied further and have more instances to be referenced upon especially in the field of refactoring.

Evaluate Refactoring Tools on More Programming Languages. As identified in RQ_2, there is a broad diversity of LLM-based refactoring tools that have been developed, yet the majority of them have only been evaluated with datasets composed primarily of either Java or Python, with limited representation of other languages. While these are both widely used in industry and software engineering research, this dominance is disproportionate to the popularity and use within industry, open-source projects, and personal development of languages such as JavaScript, Kotlin, and C++, among others. This imbalance raises questions about the general effectiveness of LLM-based refactoring

[5] https://hdsr.mitpress.mit.edu/pub/qh3dbdm9/release/2

tools, since evaluation has been largely one-dimensional. Future studies should aim to widen the scope of evaluation of these tools to enhance their effectiveness as well as the amount of developers they could benefit.

Lack of Standardized Performance Metrics for LLM-Based Refactoring. As shown in RQ_4, there are a few studies that evaluate LLMs based on established performance metrics such as Precision, Recall, and F1-score [49, 51, 52, 54], there is a noticeable lack of standardization in how the effectiveness of these tools is measured across primary studies. The evaluation metrics used in the primary studies can vary a lot, making it hard to compare different LLM-based refactoring approaches. Some use metrics such as cyclomatic complexity (CC), lines of code (LOC) to measure effectiveness [15, 57], while others rely on Accuracy rates that have very specific measurement criteria [13]. Further, some studies use manual inspection to check the quality of the refactored code [6, 16]. This inconsistency shows the need for more standardized evaluation metrics to allow more robust comparisons in LLM-based refactoring research.

Maintainability of LLM-Refactored Code. As evidenced by RQ_6, integrating LLM-based refactoring tools into existing development workflows presents challenges due to their struggle with large codebases and occasional generation of erroneous code. Additionally, their inclination to misinterpret developer intent affects not only the execution but also the maintainability of the generated code. Liu *et al.* [87] found that nearly 47% of ChatGPT-generated code snippets suffer from code style and maintainability issues–such as having variables that should be private declared as public or using implementation types instead of interface–despite a majority compiling and running without errors. This highlights the importance of developers checking the style and maintainability of LLM-generated code to ensure it meets their standards before integrating it into an existing codebase, avoiding issues down the line. Future studies should evaluate the ability of LLMs to consistently comply with stylistic choices provided in their prompts.

Integrating Human Feedback in LLM-Based Refactoring. While the refactoring tools introduced by the primary studies show the promising abilities of LLMs when it comes to refactoring, there is still concern about their reliability. As identified in RQ_6, LLMs have been observed to produce incorrect code suggestions [13] and hallucinations [5], there is a general need for manual verification of their output. To address this, adding human experts who provide feedback to create a semi-automated refactoring process can increase the reliability and consequently the effectiveness of these LLM-based tools.

Security Concerns in Using LLM-Based Refactoring Tools in Industry. With RQ_4 showing promising results with LLMs achieving strong Accuracy in refactoring tasks (with Precision scores above 74% and F1-measures reaching 87.8%), it makes sense that companies would prioritize their integration. However, more importantly, the black-box nature and proprietary risks of LLMs hold back further integration into industry. Models such as ChatGPT are closed-source, with unclear internal logic [88], therefore, it is not possible for outside businesses and users to understand how exactly refactorings are gen-

erated. Moreover, ChatGPT's input is stored and used for training, which is a significant security risk as it may contain private business information [88, 89]. This is especially concerning for companies whose systems handle sensitive personal user data such as health or financial information. As a result, future work must focus on addressing these security risks, or aiding outside entities to develop their own LLMs to keep their data protected and in-house. This could involve leveraging open-source LLMs such as CodeGen or StarCoder, which can be deployed on local, private infrastructure, eliminating data exposure risks. Researchers can also explore on-premise fine-tuning to allow models to be trained on sensitive company data without sharing it externally. Prioritizing these approaches moving forward is the most effective way for organizations to mitigate significant safety concerns when integrating LLM-based refactoring tools.

Lack of Data Leakage Prevention. As observed in our results from RQ_2, a majority of the studies concerning LLM-based refactoring, 59.61% of the primary papers, make no mention of implementing any strategies to protect against data leakage of the models they are evaluating. However, this absence of authors outlining any direct techniques for combating this data leakage may be related to the novelty of research in this field, but also the nature of these language models. In the primary papers, when refactoring tools integrate the use of LLMs, they are simply using a pre-trained model such as ChatGPT, sometimes with fine-tuning or more minimal settings manipulation (e.g.,temperature). Due to this, researchers may assume that data leakage precautions were already taken by the model provider, which is partially true, yet, depending on the evaluation data in a study, data leakage can still occur. In addition, for some LLMs, information about the full extent of training data can be unavailable due to copyright issues or just a lack of transparency [90], and can therefore make it difficult to evaluate the model on completely new data. Sallou *et al.* [76] also highlight that the model may have been previously exposed to benchmark code through pre-training, potentially inflating reported performance claims. This suggests that implicit data leakage is a significant threat to validity in LLM-based software engineering research. It does remain that a large absence of consistent reporting on data leakage prevention can hinder the reproducibility of these studies, and therefore diminishes confidence in the performance of such models.

5.1. Actionable Insights

Include Comparative Analysis Between Various LLMs Within Studies. Adding to the discussion regarding RQ_1, we believe it would be more helpful for future works to consider comparing multiple LLMs in their studies and pursuits to not only find what is the best LLM for their specific case of usage but also to help find which LLMs excel in what field and why specific LLMs are better than others for that task. This can help LLMs grow in the field of refactoring.

Further Studies on Less Well-Known Techniques and Possible Tools. For future development, for the field of code improvement with LLMs, it can benefit from more studies into using agentic AIs to further secure and better refactoring. There can be studies done on what part of software development

are agents best at and how can we improve agents in other roles to improve refactoring and development, or what structure of development does agents excel in.

Standardize Performance Metrics for LLM-Based Refactoring. Extending the insights from RQ_4, although some studies evaluate LLMs using established performance metrics like Precision, Recall, and F1-score, there remains a lack of standardization across primary studies, with many relying on varied or manual measures that make direct comparison more complicated. Future work could extensively adopt standardized reporting and evaluation statistics, including Accuracy, Precision, Recall, and F1-score, to systematically assess LLM outputs across multiple projects. Furthermore, establishing go-to community repositories of standardized LLM refactoring test cases would allow consistent, reproducible benchmarking across studies.

Establish Standard Naming Conventions for Code Smells, Code Refactoring, and Prompt Engineering; Future Works Should State Clearly Approaches. Adding on to the topic of standardization, as mentioned in regards to RQ_3 and RQ_5, there should be a more uniform way of discussing prompting techniques, code smells and code refactoring. Future work should mention by specific names what their study works on specifically, such as what prompting techniques they used, what code smells they address, and what refactoring methods they are conducting. This helps create a more uniform research environment and can aid in analysis helping furthering the field.

Address Security Concerns in Using LLM-Based Refactoring Tools in Industry. Further, while LLMs have demonstrated strong potential in automated refactoring, achieving high Precision and F1 scores that suggest feasible industry value, the integration of such tools comes with concerns around data privacy. The lack of transparency of Large Language Models like ChatGPT makes the adoption of these technologies challenging for businesses that handle regulated and proprietary information. As a result, future work must focus on addressing these security risks, or aiding outside entities to develop their own LLMs to keep their data protected and in-house. This could involve leveraging open-source LLMs such as CodeGen or StarCoder, which can be deployed on local, private infrastructure, eliminating data exposure risks. Researchers can also explore on-premise fine-tuning to allow models to be trained on sensitive company data without sharing it externally. Prioritizing these approaches moving forward is the most effective way for organizations to mitigate significant safety concerns when integrating LLM-based refactoring tools.

Integrating Human Feedback in LLM-Based Refactoring. Building on the potential of LLMs in automated refactoring, it is important to note that their reliability remains a significant concern. As reported in RQ_6, these models can generate incorrect code suggestions and hallucinations, making manual verification necessary to ensure code quality. To address this, adding human experts who provide feedback to create a semi-automated refactoring process can increase the reliability, and consequently, the effectiveness of these LLM-based tools. Future studies should explore the implementation of interactive reinforcement learning loops, in which developers validate or fix LLM suggestions,

then feed corrections back into the model using frameworks such as Hugging Face's TRL library. Additionally, code review workflows on platforms like GitHub could be improved with LLM-assisted suggestion systems that allow human verification.

Prevent Data Leakage. Another point stemming from our analysis of RQ_2 is that most studies on LLM-based refactoring do not report implementing strategies to prevent data leakage, perhaps because they rely on pre-trained models whose internal safeguards are assumed but not independently verified. A big takeaway for any future work is that authors must explicitly report the precautions they used (e.g., training/testing splits, held-out datasets, unseen repositories), even if partial, so that results across studies are more easily comparable. Going further, framing results with the context of the level of potential leakage would aid other researchers in drawing clear and accurate interpretations of the findings.

Incorporate Guidance for Targeted Refactoring. Based on our analysis in RQ_6, LLMs struggle with refactoring large projects and code files, and incorporating semantic analysis into LLM-based refactoring enhances efficiency and precision by detecting and differentiating between related and irrelevant methods when refactoring. Incorporating analysis to guide LLMs to identify refactoring opportunities allows for more LLMs to focus on refactoring relevant instances rather than hallucinating refactorings. Future work should discuss other techniques to decrease irrelevant code inputs, provide LLM's with guidance identifying refactoring opportunities, and explore how semantic similarity is measured and applied using embedding or other representations to ensure consistency across studies.

Need for Standardization in Model Classification and Identification. It is important to note that various studies differed on the categorization of different language models as LLMs, pretrained models, or Small Language Models, creating confusion for the reader. Additionally, while some papers specified the exact model or version of the LLM they were using, others referred broadly to "ChatGPT", "DeepSeek" and other models without clarification. Establishing clearer standards for model classification and identification would help improve consistency across future work.

6. Threats To Validity

Internal and Construct threats to validity: The search process for retrieving a fully representative set of literature publications for SLR may be a threat to validity. To mitigate this threat, we followed the guidelines for SLR outlined in the following: [35, 91–94]. When considering possible limitations and issues of our research, it is important to examine the tools we and our citations have used. Since our main focus is on examining the interactions between LLMs and developers, and specifically ChatGPT and developers, it is crucial to ask whether our results have been affected by the continuous development of ChatGPT.

Over time, even within the duration of this research, OpenAI has introduced GPT-4.5, and there is news from Sam Altman that ChatGPT-5 is already in development[6]. Based on the official documentation from OpenAI, ChatGPT is also updated roughly around every 2 weeks[7]. So, how exactly do we decide a research piece done on ChatGPT has been rendered completely obsolete?
Additionally, developers and researchers have noticed that while ChatGPT has become better at certain things, writing code is something it has become worse at [86]. We are not exactly sure of the implications that this has on code refactoring yet, but we have noted the difficulties that ChatGPT has on certain refactoring tasks within our own testing, specifically with markdown files and formatting. However, because of the nature of LLMs and how fast they have been developing, it is hard to decide when a research piece is truly obsolete. Within the scope of our research, we do have to consider if the increase in efficiency we saw in developer and ChatGPT interaction has come from our decisions in creating an efficient prompting format or simply from the development that OpenAI has done on ChatGPT. Another threat to validity to consider is ChatGPT-3.5's tendency to delete and translate comments when producing refactored code candidates. Shirafuji *et al.* [15] found that the comment ratio of the original code samples decreased from 2.39% to 0.89% in the generated code samples. Also, the proportion of English comments in the code went from 95.11% to 97.16%, subsequently decreasing the proportion of non-English comments. This reduction and translation can negatively impact the code's maintainability and reliability, as it removes helpful context for understanding the code while possibly creating errors through ChatGPT's unreviewed translation.

Concerning the subjectivity of the data collection and assessments of the primary studies, the primary studies were collected and then two authors independently reviewed then cross-reviewed each other's notes. To ensure complete and correct collections of data, we also asked ChatGPT-4 to check over our written evaluations and methodologies to see if there are important data points missed and added to some of our notes. This process included sending ChatGPT-4 all of the written notes of summaries with the original papers, asking if there are any more notes or topics that should be included in the summary. This usually resulted in improved grammar, added with some sentences for extra details. In total there were 16 out of 50 papers (32%) that were used in this way. At the end of each iteration, the authors met and discussed any disagreements that had arisen in interpretations and understanding to reach a consensus.

External threats to validity: The collected papers mostly consist of academic studies, providing a solid foundation for drawing conclusions relevant to academic research. Although we cannot claim that the same methods for LLM-driven refactoring detection and execution are employed in industry. Moreover, our conclusions are within the realm of software refactoring, therefore, we cannot

[6] [https://x.com/sama/status/1889755723078443244]
[7] [https://help.openai.com/en/articles/6825453-chatgpt-release-notes]

generalize our findings beyond this field.

7. Conclusion

In this paper, we map and review the body of knowledge on LLM-based refactoring tools and approaches within software engineering. We systematically reviewed 50 papers and classified them. Our findings show that (i) LLM-driven refactoring studies utilized various different external frameworks and tools to aid in studying and enhancing LLM abilities in refactoring; (ii) the datasets and benchmarks used to evaluate LLMs for refactoring tasks vary greatly in both their composition and scale, making their standardization for more consistent comparison difficult; (iii) numerous prompt techniques were used for refactoring tasks, with Few-Shot and Context-Specific being the most common; (iv) Accuracy rates of the tools were consistently high (above 87.8%), with strong F1-measure (above 74%) and Recall (mainly above 84.3%), yet Precision varied depending on the context of each study; (v) Extract Method is the most popular and promising refactoring type employed by LLMs with high Accuracy rates and qualifying, quantifying, comparative, and correctness data points; and (vi) LLMs sometimes produce erroneous code, struggle with complex refactoring tasks, and misunderstand developers' intent. This research provides the community with information about the emerging LLM-based refactoring approaches to guide future LLM-based refactoring tools. Future work includes standardizing refactoring method terminology, assessing optimal prompt technique combinations, and expanding the evaluation of LLM-based refactoring to less prevalent programming languages.

References

[1] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[2] E. A. AlOmar, M. W. Mkaouer, C. Newman, A. Ouni, On preserving the behavior in software refactoring: A systematic mapping study, Information and Software Technology 140 (2021) 106675.

[3] I. Yanakiev, B.-M. Lazar, A. Capiluppi, Applying solid principles for the refactoring of legacy code: An experience report, Journal of Systems and Software 220 (2025) 112254.

[4] A. A. B. Baqais, M. Alshayeb, Automatic software refactoring: a systematic literature review, Software Quality Journal 28 (2) (2020) 459–502.

[5] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, D. Dig, Em-assist: Safe automated extractmethod refactoring with llms, in: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, 2024, pp. 582–586.

[6] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, T. Bryksin, D. Dig, Next-generation refactoring: Combining llm insights and ide capabilities for extract method, in: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2024, pp. 275–287.

[7] M. Dilhara, A. Bellur, T. Bryksin, D. Dig, Unprecedented code change automation: The fusion of llms and transformation by example, Proceedings of the ACM on Software Engineering 1 (FSE) (2024) 631–653.

[8] J. Cordeiro, S. Noei, Y. Zou, An empirical study on the code refactoring capability of large language models, arXiv preprint arXiv:2411.02320.

[9] Y. Gao, X. Hu, X. Yang, X. Xia, Automated unit test refactoring, Proceedings of the ACM on Software Engineering 2 (FSE) (2025) 713–733.

[10] H. Wang, Z. Xu, S. H. Tan, Testing refactoring engine via historical bug report driven llm, in: 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), IEEE, 2025, pp. 113–124.

[11] I. Palit, T. Sharma, Reinforcement learning vs supervised learning: A tug of war to generate refactored code accurately, in: 2025 International Conference on Evaluation and Assessment in Software Engineering (EASE), 2025.

[12] H. Guenoune, A. Gutierrez, M. Huchard, M. Lafourcade, P. Martin, A. Miralles, H. Zhang, Llm-assisted relational concept analysis for class model restructuring, in: International Joint Conference on Conceptual Knowledge Structures, Springer, 2025, pp. 107–123.

[13] J. Gehring, Deterministic automatic refactoring at scale, in: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2023, pp. 541–546.

[14] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, D. C. Schmidt, A prompt pattern catalog to enhance prompt engineering with chatgpt, arXiv preprint arXiv:2302.11382.

[15] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, Y. Watanobe, Refactoring programs using large language models with few-shot examples, in: 2023 30th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2023, pp. 151–160.

[16] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, E. A. AlOmar, Exploring chatgpt's code refactoring capabilities: An empirical study, Expert Systems with Applications 249 (2024) 123602.

[17] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. Newman, A. Ouni, How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations, in: Proceedings of the 21st International Conference on Mining Software Repositories, 2024, pp. 202–206.

[18] O. S. Chavan, D. D. Hinge, S. S. Deo, Y. Wang, M. W. Mkaouer, Analyzing developer-chatgpt conversations for software refactoring: An exploratory study, in: Proceedings of the 21st International Conference on Mining Software Repositories, 2024, pp. 207–211.

[19] S. Singh, S. Kaur, A systematic literature review: Refactoring for disclosing code smells in object oriented software, Ain Shams Engineering Journal 9 (4) (2018) 2129–2151.

[20] J. Al Dallal, A. Abdin, Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review, IEEE Transactions on Software Engineering 44 (1) (2017) 44–69.

[21] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, Journal of Software Maintenance and Evolution: research and practice 23 (3) (2011) 179–202.

[22] E. A. AlOmar, M. W. Mkaouer, A. Ouni, Behind the intent of extract method refactoring: A systematic literature review, IEEE Transactions on Software Engineering 50 (4) (2024) 668–694.

[23] J. Jiang, F. Wang, J. Shen, S. Kim, S. Kim, A survey on large language models for code generation, ACM Trans. Softw. Eng. Methodol.Just Accepted.

[24] J. He, C. Treude, D. Lo, Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead, ACM Transactions on Software Engineering and Methodology 34 (5) (2025) 1–30.

[25] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large language models for software engineering: A systematic literature review, ACM Transactions on Software Engineering and Methodology 33 (8) (2024) 1–79.

[26] D. M. Zapkus, A. Slotkienė, Unit test generation using large language models: A systematic literature review, Lietuvos magistrantų informatikos ir IT tyrimai: konferencijos darbai, 2024 m. gegužės 10 d. (2024) 136–144.

[27] J. Wang, Y. Chen, A review on code generation with llms: Application and evaluation, in: 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI), IEEE, 2023, pp. 284–289.

[28] R. A. Husein, H. Aburajouh, C. Catal, Large language models for code completion: A systematic literature review, Computer Standards & Interfaces 92 (2025) 103917.

[29] Y. Xu, F. Lin, J. Yang, N. Tsantalis, et al., Mantra: Enhancing automated method-level refactoring with contextual rag and multi-agent llm collaboration, arXiv preprint arXiv:2503.14340.

[30] F. Batole, A. Bellur, M. Dilhara, M. R. Ullah, Y. Zharov, T. Bryksin, K. Ishikawa, H. Chen, M. Morimoto, S. Motoura, et al., Together we are better: Llm, ide and semantic embedding to assist move method refactoring, in: 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2025.

[31] R. Gheyi, M. Ribeiro, J. Oliveira, Evaluating the effectiveness of small language models in detecting refactoring bugs, arXiv preprint arXiv:2502.18454.

[32] I. Palit, G. Shetty, H. Arif, T. Sharma, Automatic refactoring candidate identification leveraging effective code representation, in: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2023, pp. 369–374.

[33] A. Alazba, H. Aljamaan, M. Alshayeb, Smellybot: An ai-powered software bot for code smell detection, Software: Practice and Experience.

[34] H. Liu, Y. Wang, Z. Wei, Y. Xu, J. Wang, H. Li, R. Ji, 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, 2023, pp. 740–752.

[35] B. Kitchenham, S. Charters, et al., Guidelines for performing systematic literature reviews in software engineering (2007).

[36] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, M. A. Babar, Understanding and addressing quality attributes of microservices architecture: A systematic literature review, Information and software technology 131 (2021) 106449.

[37] T. Dybå, T. Dingsøyr, Empirical studies of agile software development: A systematic review, Information and software technology 50 (9-10) (2008) 833–859.

[38] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, H. Liu, Exploring the potential of general purpose llms in automated software refactoring: an empirical study, Automated Software Engineering 32 (1) (2025) 26.

[39] Z. Zhang, Z. Xing, X. Ren, Q. Lu, X. Xu, Refactoring to pythonic idioms: A hybrid knowledge-driven approach leveraging large language models, Proceedings of the ACM on Software Engineering 1 (FSE) (2024) 1107–1128.

[40] N. Baumgartner, P. Iyenghar, T. Schoemaker, E. Pulvermüller, Ai-driven refactoring: A pipeline for identifying and correcting data clumps in git repositories, Electronics 13 (9) (2024) 1644.

[41] Y. Zhang, Y. Li, G. Meredith, K. Zheng, X. Li, Move method refactoring recommendation based on deep learning and llm-generated information, Information Sciences 697 (2025) 121753.

[42] N. Alturayeif, J. Hassine, Refactoring goal-oriented models: a linguistic improvement using large language models, Software and Systems Modeling (2025) 1–29.

[43] D.-K. Kim, Comparative analysis of design pattern implementation validity in llm-based code refactoring, Journal of Systems and Software 230 (2025) 112519.

[44] H. Wang, Z. Xu, H. Zhang, N. Tsantalis, S. H. Tan, Towards understanding refactoring engine bugs, ACM Transactions on Software Engineering and Methodology.

[45] Z. Wang, T. He, R. Zhao, T. Zheng, Exploration and improvement of capabilities of llms in code refinement task., International Journal of Software & Informatics 15 (2).

[46] A. Midolo, E. Tramontana, Refactoring loops in the era of llms: A comprehensive study, Future Internet 17 (9) (2025) 418.

[47] K. Xu, G. L. Zhang, X. Yin, C. Zhuo, U. Schlichtmann, B. Li, Hlsrewriter: Efficient refactoring and optimization of c/c++ code with llms for high-level synthesis, ACM Transactions on Design Automation of Electronic Systems.

[48] A. Tkachuk, Determining the capabilities of generative artificial intelligence tools to increase the efficiency of refactoring process, Technology audit and production reserves 3 (83).

[49] A. Ghammam, D. E. Rzig, M. Almukhtar, R. Khalsi, F. Hassan, M. Kessentini, Build code needs maintenance too: A study on refactoring and technical debt in build systems, in: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), IEEE, 2025, pp. 616–628.

[50] E. Ksontini, M. Mastouri, R. Khalsi, W. Kessentini, Refactoring for dockerfile quality: A dive into developer practices and automation potential, in: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), IEEE, 2025, pp. 788–800.

[51] D. Cui, J. Wang, Q. Wang, P. Ji, M. Qiao, Y. Zhao, J. Hu, L. Wang, Q. Li, Three heads are better than one: Suggesting move method refactoring opportunities with inter-class code entity dependency enhanced hybrid hypergraph neural network, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 745–757.

[52] D. Cui, Q. Wang, Y. Zhao, J. Wang, M. Wei, J. Hu, L. Wang, Q. Li, One-to-one or one-to-many? suggesting extract class refactoring opportunities with intra-class dependency hypergraph neural network, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 1529–1540.

[53] X. Gao, Y. Xiong, D. Wang, Z. Guan, Z. Shi, H. Wang, S. Li, Preference-guided refactored tuning for retrieval augmented code generation, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 65–77.

[54] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, L. Zhang, ismell: Assembling llms with expert toolsets for code smell detection and refactoring, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1345–1357.

[55] B. Zhang, P. Liang, Q. Feng, Y. Fu, Z. Li, Copilot-in-the-loop: Fixing code smells in copilot-generated python code using copilot, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 2230–2234.

[56] F. Batole, A. Bellur, M. Dilhara, M. R. Ullah, Y. Zharov, T. Bryksin, K. Ishikawa, H. Chen, M. Morimoto, S. Motoura, et al., Leveraging llms, ides, and semantic embeddings for automated move method refactoring, in: 2025 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2025.

[57] J. Choi, G. An, S. Yoo, Iterative refactoring of real-world open-source programs with large language models, in: International Symposium on Search Based Software Engineering, Springer, 2024, pp. 49–55.

[58] J. White, S. Hays, Q. Fu, J. Spencer-Smith, D. C. Schmidt, Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, in: Generative AI for Effective Software Development, Springer, 2024, pp. 71–108.

[59] R. Ishizue, K. Sakamoto, H. Washizaki, Y. Fukazawa, Improved program repair methods using refactoring with gpt models, in: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, 2024, pp. 569–575.

[60] A. Ben Mrad, A. M. O. Thiombiano, M. W. Mkaouer, B. Hnich, Assessing large language models effectiveness in outdated method renaming, in: International Conference on Service-Oriented Computing, Springer, 2024, pp. 253–260.

[61] A. Midolo, M. Di Penta, Automated refactoring of non-idiomatic python code: A differentiated replication with llms, in: 2025 IEEE/ACM International Conference on Program Comprehension (ICPC), 2025.

[62] A. Menolli, B. Strik, L. Rodrigues, Teaching refactoring to improve code quality with chatgpt: An experience report in undergraduate lessons, in: Proceedings of the XXIII Brazilian Symposium on Software Quality, 2024, pp. 563–574.

[63] L. Wang, Q. Wang, J. Wang, Y. Zhao, M. Wei, Z. Quan, D. Cui, Q. Li, Hecs: A hypergraph learning-based system for detecting extract class refactoring opportunities, in: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 1851–1855.

[64] S. Divyansh, P. Appoorva, S. Singh, A. Ershadi, H. Makwana, E. AlOmar, Evaluating the effectiveness of chatgpt in improving code quality, in: 2025 IEEE Evaluating the Effectiveness of ChatGPT in Improving Code Quality (ICMI), IEEE, 2025.

[65] C. Dong, Y. Jiang, Y. Zhang, Y. Zhang, L. Hui, Chatgpt-based test generation for refactoring engines enhanced by feature analysis on examples, in: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE Computer Society, 2025, pp. 746–746.

[66] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, X. Peng, Exploring the potential of chatgpt in automated code refinement: An empirical study, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.

[67] Q. Guo, X. Xie, S. Liu, M. Hu, X. Li, L. Bu, Intention is all you need: Refining your code from your intention., 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) (2025) 728–728.

[68] V. Rajendran, D. Besiahgari, S. C. Patil, M. Chandrashekaraiah, V. Challagulla, A multi-agent llm environment for software design and refactoring: A conceptual framework, in: SoutheastCon 2025, IEEE, 2025, pp. 488–493.

[69] E. A. AlOmar, L. Xu, S. Martinez, A. Peruma, M. W. Mkaouer, C. D. Newman, A. Ouni, Chatgpt for code refactoring: Analyzing topics, interaction, and effective prompts, in: International Conference on Collaborative Advances in Software and ComputiNg (CASCON), 2025.

[70] G. Caumartin, Q. Qin, S. Chatragadda, J. Panjrolia, H. Li, D. E. Costa, Exploring the potential of llama models in automated code refinement: A replication study, in: 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2025, pp. 681–692.

[71] G. Pandini, A. Martini, A. N. Videsjorden, F. A. Fontana, An exploratory study on architectural smell refactoring using large languages models, in: 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C), IEEE, 2025, pp. 462–471.

[72] G. Amaral, H. Gomes, E. Figueiredo, C. Bezerra, L. Rocha, Improving javascript test quality with large language models: Lessons from test smell refactoring, in: Simpósio Brasileiro de Engenharia de Software (SBES), SBC, 2025, pp. 776–782.

[73] A. P. Pucho, A. M. Ferreira, E. J. R. Cirilo, B. B. Cafeo, Refactoring python code with llm-based multi-agent systems: An empirical study in ml software projects, in: Simpósio Brasileiro de Engenharia de Software (SBES), SBC, 2025, pp. 678–684.

[74] V. F. de Sousa, C. de Souza Baptista, A. L. F. Alves, H. F. de Figueirêdo, Effectiveness of small and large language models for pl/sql bad smell detection, in: Simpósio Brasileiro de Banco de Dados (SBBD), SBC, 2025, pp. 399–412.

[75] D. Gautam, S. Garg, J. Jang, N. Sundaresan, R. Z. Moghaddam, Refactorbench: Evaluating stateful reasoning in language agents through code, in: NeurIPS 2024 Workshop on Open-World Agents, 2024.

[76] J. Sallou, T. Durieux, A. Panichella, Breaking the silence: the threats of using llms in software engineering, in: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2024, pp. 102–106.

[77] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, Y. Liu, The scope of chatgpt in software engineering: A thorough investigation, arXiv preprint arXiv:2305.12138.

[78] T. Xiao, C. Treude, H. Hata, K. Matsumoto, Devgpt: Studying developer-chatgpt conversations, in: 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), IEEE, 2024, pp. 227–230.

[79] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, A. De Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 482–482.

[80] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empirical Software Engineering 21 (2016) 1143–1191.

[81] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change-and fault-proneness, Empirical Software Engineering 17 (2012) 243–275.

[82] A. Peruma, A preliminary study of android refactorings, in: 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2019, pp. 148–149. doi:10.1109/MOBILESoft.2019.00030.

[83] A. Birillo, I. Vlasov, A. Burylov, V. Selishchev, A. Goncharov, E. Tikhomirova, N. Vyahhi, T. Bryksin, Hyperstyle: A tool for assessing the code quality of solutions to programming assignments, in: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1, 2022, pp. 307–313.

[84] D. Thomas, Synthesisable recursion for c++ hls tools, 2016, pp. 91–98. doi:10.1109/ASAP.2016.7760777.

[85] F. Vernier, A. Miralles, F. Pinet, N. Carluer, V. Gouy, G. Molla, K. Petit, Eis pesticides: An environmental information system to characterize agricultural activities and calculate agro-environmental indicators at embedded watershed scales, Agricultural Systems 122 (2013) 11–21.

[86] L. Chen, M. Zaharia, J. Zou, How is chatgpt's behavior changing over time?, Harvard Data Science Review 6 (2).

[87] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, D. Lo, Refining chatgpt-generated code: Characterizing and mitigating code quality issues, ACM Transactions on Software Engineering and Methodology 33 (5) (2024) 1–26.

[88] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, et al., On the opportunities and risks of foundation models, arXiv preprint arXiv:2108.07258.

[89] H. Kibriya, W. Z. Khan, A. Siddiqa, M. K. Khan, Privacy issues in large language models: a survey, Computers and Electrical Engineering 120 (2024) 109698.

[90] A. Buick, Copyright and ai training data—transparency to the rescue?, Journal of Intellectual Property Law and Practice 20 (3) (2025) 182–192.

[91] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th international conference on evaluation and assessment in software engineering, 2014, pp. 1–10.

[92] B. Kitchenham, Procedures for performing systematic reviews, Keele, UK, Keele University 33 (2004) (2004) 1–26.

[93] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, M. Khalil, Lessons from applying the systematic literature review process within the software engineering domain, Journal of systems and software 80 (4) (2007) 571–583.

[94] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering–a systematic literature review, Information and software technology 51 (1) (2009) 7–15.