# An Evaluation of GPT-4.1 in Assisting Java Code Refactoring

Henrique Silva
Federal University of Pará
Belém, Brazil
jose.henrique.silva@icen.ufpa.br

Cleidson R. B. de Souza
Federal University of Pará
Belém, Brazil
cleidson.desouza@acm.org

## Abstract

Code refactoring is an activity widely used in the daily practice of software development with the goal of making a codebase maintainable and enabling the development of new features more efficiently. Since the release of ChatGPT, LLMs have shown great potential for performing diverse software engineering tasks. This study aims to evaluate the performance of GPT-4.1 in carrying out refactorings. To do so, we employ three different metrics: validity, correctness, and normalized edit distance to assess five different types of refactoring, namely: extract method, extract variable, rename attribute, rename method, and rename parameter. Our results suggest that GPT-4.1 produces refactorings with high validity and the generated code is textually close to the reference (low normalized edit distance), but its correctness underperforms. We conclude that, with the prompts used here, GPT-4.1 does not reliably produce correct refactorings, and its outputs should always be reviewed (by running unit tests or checking behavior manually) before being trusted.

## Keywords

Code Refactoring, Large Language Models, GPT-4.1, Software Engineering, Empirical Evaluation, Java

## 1 Introduction

Since the release of ChatGPT by OpenAI in November 2022, Large Language Models (LLMs) have become popular tools for a variety of tasks, with great potential across software engineering activities such as coding, design, requirements, repair, refactoring, performance improvement, documentation, and data analysis [4].

Several studies show how LLMs can support these software development activities. For example, Mu et al. [9] discuss code generation, while Wu et al. [13] explore fault localization to enable automated debugging. Xia and Zhang [14] address code repair in a fully conversational manner, and Piya and Sullivan [10] cover code and test generation following the test-driven development process.

Code refactoring is a common task in the software development cycle, because a well-kept codebase remains maintainable over time. Within this context of using LLMs for software engineering activities, this study evaluates how GPT-4.1 assists software engineers in carrying out five refactoring types: *extract method*, *extract variable*, *rename attribute*, *rename method*, and *rename parameter*.

To perform the evaluation, we use the dataset created by Liu et al. [7], which provides prompts that follow prompt-engineering best practices to improve the LLM's responses and includes 180 real refactorings from 20 open-source projects. This dataset consists of source code before and after the application of a single refactoring type. We use the "before" code to build the prompts and the "after" code as a reference for the normalized edit distance metric. In addition, we extract two further metrics: validity and correctness.

Our results suggest that GPT-4.1 performs several refactorings with high validity that are textually close to the reference, but its correctness is lower: some responses compile yet introduce functional changes. Therefore, developers should always review the output.

## 2 Background

This section explains the foundational concepts needed for a correct understanding of our study.

### 2.1 Refactoring

Refactoring means changing a software system's internal structure to make it easier to understand and cheaper to maintain, without changing its observable behavior. This practice improves the software's design, makes the code more comprehensible and easier to debug, and thereby speeds up the development of new features [5].

Fowler [5] lists and explains 61 refactorings. However, this list is not exhaustive. Our study's scope is limited to a subset of these types of refactoring, presented in Subsection 2.2.

### 2.2 Types of Refactoring

This subsection details which refactorings we study: *extract method*, *extract variable*, *rename attribute*, *rename method*, and *rename parameter*. A brief description of each is given below.

*Extract method* turns a loose or repeated code fragment into a well-named function used in the fragment's place, so the name conveys the fragment's purpose without the reader needing its implementation. Similarly, *extract variable* breaks complex or repeated expressions into named intermediate variables that clarify what each produces. Turning from code fragments to names, *rename attribute* gives a class's fields more descriptive names. Likewise, *rename method* gives a method a name that states plainly what it does, useful because methods are the units joined to build the system. Finally, *rename parameter* renames a method's parameters to make explicit how the function should be used [5].

Having covered the technical concepts needed to understand our study, we now present our methodology.

## 3 Related Work

This section groups studies that apply LLMs to software refactoring and positions our contribution against them.

AlOmar et al. [2] characterized real developer-ChatGPT conversations without measuring correctness. In addition, Pomian et al. [11] combined LLM suggestions with IDE support for extract method, arguing LLMs alone are unsafe. Likewise, Midolo and Di Penta [8] replicated this line of work on Python code, reporting frequent syntactic errors and behavioral changes. These works motivate our use of validity and correctness as core metrics.

Finally, Liu et al. [7] evaluated general-purpose LLMs on refactoring and released the dataset we reuse. We differ by evaluating a newer model, GPT-4.1, on five refactoring types, and by assessing how well refactorings are executed (through validity, correctness, and normalized edit distance) rather than whether they are identified.

## 4 Methodology

This section presents the entire development process of our study, detailing the procedure used to obtain the results. The methodology and the data used in our study are inspired by the work of Liu et al. [7].

### 4.1 Data Collection

The data used in our study are part of the dataset built by Liu et al. [7]. This dataset, called *ref-Dataset*, is built from 20 popular Java projects chosen by Grund et al. [6]. These projects stand out because they have a rich evolutionary history that began in the 2000s and come from different application domains (such as code analysis and search engines), which helps reduce bias in the evaluations [7]. Moreover, they are publicly available, making any evaluation easy to access and reproduce.

We now explain how that dataset was built. Since refactorings in real projects are not performed in isolation and are usually mixed with feature changes, the code extracted from the repositories was processed to retain only the changes related to refactoring. As a result, the "before" code in the dataset contains only refactoring opportunities, while the "after" code is the code with the refactoring applied.

The prompts we use are also the same as those of Liu et al. [7]. They developed different versions of prompts that follow best practices [1, 3]. However, in our study, we use only one of these versions. It specifies the role the LLM should take when performing the refactoring; the code snippet to which the refactoring should be applied (in our study, the complete code of the classes); and which refactoring should be done, together with a brief explanation of its objective in modifying the code.

What distinguishes this version of the prompt from the others is that it explicitly states which type of refactoring should be performed, which improves the LLM's performance in identifying refactorings [7].

### 4.2 Types of Refactoring Used

The choice to analyze the five aforementioned refactorings is due primarily to two findings of Liu et al. [7]: that *rename attribute*, *rename method*, *rename parameter*, and other renaming refactorings have a refactoring-opportunity identification success rate of 67.5%; and that 70% of the solutions the LLMs proposed for *extract method*, *extract variable*, and other extraction refactorings are of the same quality as, or better than, those produced by developers.

Furthermore, each of these five types of refactoring has 20 refactoring examples in the dataset, and each example consists of two samples: one represents the code before the refactoring is applied and the other after it is applied. We use the sample prior to the refactoring in the prompt.

We run each prompt five times to measure the model's run to run variability, resulting in 500 responses (Figure 1). The LLM used in this evaluation is one of OpenAI's models, specifically the GPT-4.1 model, dated April 14, 2025. We chose this model because OpenAI introduced it as optimized for coding tasks and it was the most recent in its line at the time.

All prompts are submitted directly to OpenAI's Application Programming Interface (API) with temperature = 1, top_p = 1, and no context shared between calls, through the Software Development Kit (SDK) for Python. Both the prompts and the responses are stored in a GitHub repository (Figure 1).

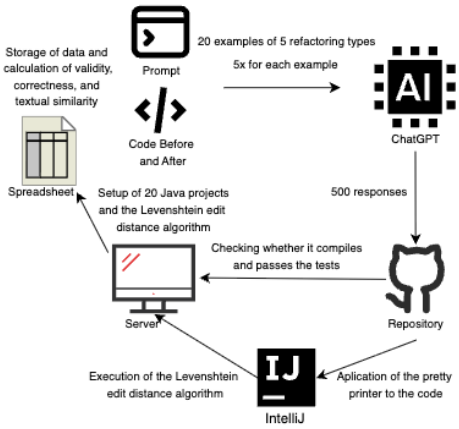


**Figure 1: Macro representation of the methodological process.**

### 4.3 Metrics Used

From these responses, we compute validity, correctness, and normalized edit distance metrics for subsequent data analysis. To obtain the validity and correctness data, we clone and configure the 20 open-source projects on a local server to run the unit tests when present (Figure 1). Further, we check out the specific commit of the project — from which the samples forming the dataset are extracted — using the *git checkout* command, in order to verify validity and correctness.

*Validity* consists of checking whether the refactored code produced by the LLM compiles. That is, whether substituting the LLM-generated code for the project's original code would produce no error that prevents the project from running. For the subsequent calculation of the metric, responses that compile are marked 1 and those that do not are marked 0. In addition, one example each for *rename method* and *rename parameter* could not be evaluated. Hence, those two types use 19 examples instead of 20 (Figure 2).

*Correctness*, in turn, consists of running the unit tests of the corresponding class that underwent the refactoring. Thus, the metric is obtained by dividing the number of tests that passed by the total number of tests. Correctness requires unit tests, which not all examples have. Examples without tests are excluded from the correctness averages, leaving 7 to 14 examples per type (Figure 3). Because refactoring must preserve behavior, a behavioral change makes tests fail. Correctness, therefore, captures behavior preservation.

Beyond validity and correctness, we measure textual closeness with the *normalized edit distance*: the Levenshtein edit distance (LED) – the minimum number of character insertions, deletions, and substitutions needed to turn the generated code into the reference code – divided by the number of characters in the reference code, expressed as a percentage. A lower value means the generated code is closer to the reference, and 0% means they are identical. We compute the Levenshtein distance using the *edit distance* function of the NLTK Python library, after formatting both the generated and reference code with IntelliJ IDEA 2024.2.6 (Community Edition)'s formatter so that formatting differences do not inflate the distance.

## 4.4 Experimental Setup

The model's responses contained explanatory text, so we extracted the refactored class by keeping only the Java snippet. We placed that class in the original project at the commit the sample was drawn from, installed the required dependencies, and built each project with Maven or Gradle using its required Java version. We then compiled and ran the unit tests manually through IntelliJ's built-in test runner, with no timeout on test execution.

## 4.5 Data Analysis

To discover patterns and relationships in the available data, we summarize each metric with a descriptive analysis (maximum, minimum, mean, and standard deviation), aggregated in two stages. First, for each example, we average the metric over its five GPT-4.1 responses: the percentage of responses that compile (validity), the percentage of unit tests that pass (correctness), and the normalized edit distance to the reference refactoring. Then, for each refactoring type, we compute the mean and standard deviation of these per-example values, obtaining a single value per metric per type. For validity and correctness metrics, not all 20 examples were considered (see Figures 2, 3, and 4). Finally, we present these distributions as boxplots, one per metric across the five refactoring types.

# 5 Results

This section presents the results obtained from the metrics computed according to the methodology specified in the previous section.

## 5.1 Validity

Figure 2 and Table 1 show most responses compile. *Extract method* is the clear outlier, with the lowest mean and by far the widest spread. The other types compile far more reliably; among the renaming types, *rename method* and *rename parameter* compile in all five runs for 68.42% and 73.68% of examples, and *rename attribute* is the most stable, with most examples at 100%.

## 5.2 Correctness

Correctness is the most dispersed metric for every type (Figure 3), with *rename attribute* the widest, ranging from 0% to 100%. As Table 1 shows, the share of examples that compiled and passed every test in all five runs ranges from 30% (*extract variable*) to 63.63% (*rename parameter*), so even when tests exist, consistent behavior preservation is far from guaranteed.

## 5.3 Normalized Edit Distance

Generated code is textually close to the reference (Figure 4). *Extract method* requires the most change, with a mean of 16.5% and two outliers at 42% and 49%, whereas the renaming types stay below 10%, often near 1–3%.

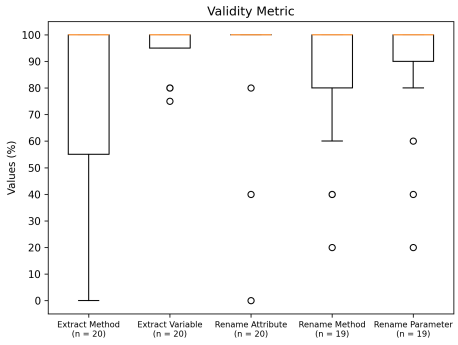


**Figure 2: Comparison of the validity metric across the five refactoring types.**

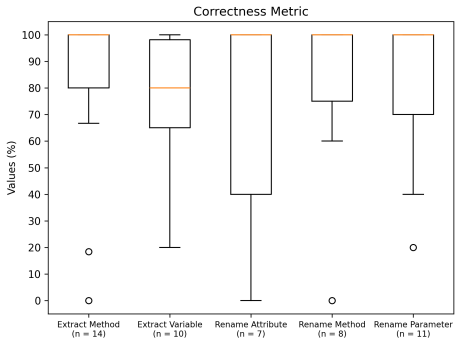


**Figure 3: Comparison of the correctness metric across the five refactoring types. Correctness is computed only over the examples that have unit tests, so the number of examples varies across types.**

## 5.4 Aggregated Results

Table 1 reports the mean and standard deviation of each metric per refactoring type: *extract method* (EM), *extract variable* (EV), *rename attribute* (RA), *rename method* (RM), and *rename parameter* (RP). The *Mean* column averages the five per-type means, (EM + EV + RA + RM + RP)/5: on average, 86.88% of the generated code is valid, 77.85% is correct, and 6.73% of characters must change to match the reference. The three bottom rows report, per type, the number of examples that compiled in all five runs (see Figure 2 for the number of examples), the percentage that also passed every test in all five runs (among examples with unit tests), and the number of examples carrying unit tests. Correctness and the pass rate are computed only over those examples.

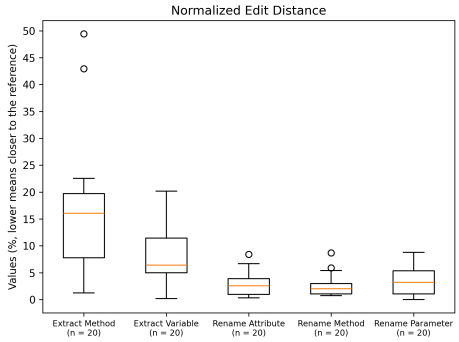


**Figure 4: Comparison of the normalized edit distance metric across the five refactoring types.**

**Table 1: Mean and standard deviation (SD) of each metric for each refactoring type, share of responses that compile in all runs and that compile and pass all tests, and the number of examples per type that carry unit tests**

| Metric | Statistic | EM | EV | RA | RM | RP | Mean |
|---|---|---|---|---|---|---|---|
| Validity | Mean | 75% | 94.75% | 91.00% | 85.26% | 88.42% | 86.88% |
| | SD | 35.46% | 9.38% | 25.52% | 25.68% | 23.39% | 23.88% |
| Correctness | Mean | 81.64% | 77.27% | 68.57% | 80.00% | 81.81% | 77.85% |
| | SD | 32.65% | 25.13% | 47.40% | 35.45% | 28.91% | 33.90% |
| Normalized Edit Distance | Mean | 16.50% | 8.35% | 2.99% | 2.55% | 3.30% | 6.73% |
| | SD | 12.08% | 5.56% | 2.32% | 2.10% | 2.65% | 4.94% |
| Compiled in all runs (5/5 runs) | — | 55% | 75% | 85% | 68.42% | 73.68% | 71.42% |
| Compiled and passed all tests (5/5 runs) | — | 57.14% | 30.00% | 57.14% | 62.50% | 63.63% | 54.08% |
| Examples with unit tests (n) | — | 14 | 10 | 7 | 8 | 11 | 10 |

## 6 Discussion

Table 1 shows the LLM performs better on validity and normalized edit distance than on correctness. That is, the correctness metric remains below the others. The exception is the *extract method* refactoring, which deviates from this pattern because its validity and correctness metrics have very close values.

These results indicate that the valid responses of LLMs cannot be trusted: even when free of compilation errors and textually similar, the generated code can change the behavior relative to the original code, which may introduce bugs or even alter business rules. These results align with the literature, which suggests refactorings carried out by LLMs are unsafe since they change program behavior, introduce syntax errors, or hallucinate [7, 8, 11].

Thus, unit tests must be run when they exist, or the behavior must be checked manually to confirm it was not changed. Therefore, because refactoring means not modifying the code's behavior, and GPT-4.1 does modify that behavior, GPT-4.1 with the prompts used in this paper does not reliably produce correct refactorings without verification.

Beyond the per-metric means, the five runs frequently disagreed for the same example. For compilation, the share of unstable examples — those that compiled in some but not all five runs — ranged from 10.00% (rename attribute) to 35.00% (extract method) across refactoring types. For the tests-passing outcome, computed only over the examples that carry unit tests (7 to 14 per type; see Table 1), the unstable share ranged from 14.28% (rename attribute, 1 of 7) to 70.00% (extract variable, 7 of 10). In other words, a single call to GPT-4.1 is often not representative of its behavior on the same input, which reinforces that its output must be verified rather than trusted from one run.

In addition, renaming refactorings outperform extraction ones. Their boxplots are tighter and sit nearer the top for validity and correctness, and nearer the bottom for normalized edit distance (Figures 2, 3, and 4), consistent with Table 1. Liu et al. [7] observed the same trend, though they measured refactoring-opportunity identification rather than execution.

## 7 Threats to Validity

*Construct validity*: normalized edit distance compares plain text and may not capture refactoring quality, and the test-pass rate may not fully reflect behavior preservation since we cannot guarantee the test suites are high-quality. *Internal validity*: we use a single prompt version that names one refactoring type, and the input code is pre-trimmed to the refactoring opportunity. *External validity*: we cover only Java, five refactoring types, and one LLM, which limits generalization to other languages, models, and refactoring. *Conclusion validity*: our study has a small sample per refactoring type (20 examples), and we rely only on descriptive statistics, without significance testing.

## 8 Conclusion and Future Work

Our study evaluated GPT-4.1's performance on five refactorings (*extract method*, *extract variable*, *rename attribute*, *rename method*, and *rename parameter*) through the validity, correctness, and normalized edit distance metrics.

Although GPT-4.1 produces many valid and textually similar refactorings (with renaming refactorings outperforming extraction ones), this does not always mean the code is correct because it can introduce semantic changes that cause unit tests to fail. With the prompts used here, the model therefore does not reliably produce correct refactorings, and its output must always be reviewed. The LLM should be seen as an ally in these tasks, not a replacement for verification.

For future work, we see two directions: (i) using only examples that have tests, or generating the missing tests with a tool such as *SafeRefactor* [12], which automatically produces tests that detect behavior changes, giving more accurate correctness metrics; and (ii) evaluating LLMs in more realistic settings, since this study's prompts named the refactoring, requested a single type, and used code pre-trimmed to the relevant opportunities, whereas real-world prompts are far more generic [2].

## Artifact Availability

All prompts, responses, dataset, and results are available in our replication package at https://github.com/jhenriquedsilva/llms-refactoring.

## Acknowledgements

We thank CNPq (grants 420406/2023-9 and 308101/2025-1) for partially funding this research. GenAI tools were used both as subjects of the empirical investigation and as writing support: Claude assisted with grammar checking, textual cohesion, and LaTeX table formatting. The authors reviewed all content and take full responsibility for it.

## References


[1] Fatih Kadir Akın. 2024. Awesome ChatGPT Prompts. https://github.com/f/awesome-chatgpt-prompts. Accessed: 2025-08-31.

[2] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. 2024. How to Refactor this Code? An Exploratory Study on Developer-ChatGPT Refactoring Conversations. In *Proceedings of the 21st IEEE/ACM International Conference on Mining Software Repositories (MSR 2024)*. ACM / IEEE, Lisbon, Portugal, 202–206. doi:10.1145/3643991.3645081

[3] Dair.AI. 2024. Prompt Engineering Guide. https://github.com/dair-ai/Prompt-Engineering-Guide/blob/main/guides/prompts-intro.md. Accessed: 2025-08-31.

[4] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, Los Alamitos, USA, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[5] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Professional.

[6] Felix Grund, Shaiful Alam Chowdhury, Nick C Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE '21)*. IEEE, Madrid, Spain, 1510–1522. doi:10.1109/ICSE43902.2021.00135

[7] Bo Liu, Yanjie Jiang, Yuxia Zhang, Nan Niu, Guangjie Li, and Hui Liu. 2025. Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study. *Automated Software Engineering* 32, 26 (2025). doi:10.1007/s10515-025-00500-0

[8] Alessandro Midolo and Massimiliano Di Penta. 2025. Automated Refactoring of Non-Idiomatic Python Code: A Differentiated Replication with LLMS. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE, Ottawa, Canada, 01–11. doi:10.1109/ICPC66645.2025.00020

[9] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354. doi:10.1145/3660810

[10] Sanyogita Piya and Allison Sullivan. 2024. LLM4TDD: Best practices for test driven development using large language models. In *Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code '24)*. ACM, New York, USA, 14–21. doi:10.1145/3643795.3648382

[11] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method. In *Proceedings of the 40th IEEE International Conference on Software Maintenance and Evolution (ICSME 2024)*. IEEE, Flagstaff, USA, 275–287. doi:10.1109/ICSME58944.2024.00034

[12] Gustavo Soares. 2010. Making program refactoring safer. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ACM, Cape Town, South Africa, 521–522. doi:10.1145/1810295.1810461

[13] Yonghao Wu, Zheng Li, Jie M. Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large Language Models in Fault Localisation. (2023). doi:10.48550/ARXIV.2308.15276

[14] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Vienna, Austria, 819–831. doi:10.1145/3650212.3680323