Investigating the Refactoring Capabilities of Small Open-Weight Language Models

Tamás Márton , Balázs Szalontai * , Balázs Pintér , Tibor Gregorics

Posted Date: 4 March 2026

doi: 10.20944/preprints202603.0295.v1

Keywords: refactoring; small language models; open-weight; code equivalence; code quality

 doi:10.20944/preprints202603.0295.v1

Article

Investigating the Refactoring Capabilities of Small Open-Weight Language Models

Tamás Márton, Balázs Szalontai *, Balázs Pintér and Tibor Gregorics

Department of Software Technology and Methodology, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
* Correspondence: szalontaib@inf.elte.hu

Abstract

Refactoring is essential for developing maintainable software. Using Large Language Models in software engineering is widespread, but compared to well-established domains such as code generation, reliable refactoring is still relatively underexplored. In this paper, we perform a broad analysis on the refactoring capabilities of small open-weight language models (SLMs) by evaluating 12 models on 3,453 Python programs. Our study focuses on the two defining aspects of refactoring: behavior preservation and code quality improvement. We evaluate these properties using unit tests and various code metrics. Across models ranging from 0.5B to 8B parameters, most models improve code quality. Larger models are more reliable, as they preserve behavior more consistently. Reasoning models often make more significant changes while refactoring. Allowing models to generate reasoning traces improves performance, but only for models larger than 4B. For smaller models, reasoning in fact reduces refactoring reliability. The difficulty of the underlying task affects refactoring performance, with more complex tasks associated with higher failure rates. Our results indicate that current open SLMs can support refactoring tasks, especially larger ones with reasoning capabilities, but they are best used with human oversight.

Keywords: refactoring; small language models; open-weight; code equivalence; code quality

1. Introduction

Both open and closed-weight Large Language models (LLMs) have been used for a wide variety of software engineering-related tasks, including code generation, summarization and bugfixing [1–4]. Their code refactoring capability is a less well investigated area. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [5]. Code improvements can manifest in various ways, including reducing code complexity, enhancing maintainability, and improving adherence to coding standards.

Prior work has demonstrated the potential of LLMs to perform code refactoring [6–12]. The degree to which these models can improve source code has often been measured using software quality metrics, such as Cyclomatic Complexity [13], Maintainability Index [14], Halstead metrics [15], and complience to PEP-8 guidelines [16]. Correctness of the refactored program is often validated by executing unit tests on the modified programs. Validating the correctness of refactorings can be crucial, as LLMs have been shown to introduce new bugs in certain cases [10], making them unreliable for fully automated refactoring.

Previous research on LLM refactoring capabilities has primarily focused on large, closed-weight models. This leaves smaller, open-weight models underrepresented, with only a few studies exploring them [7,9,11]. Moreover, studies typically examine only a narrow range of models (often only one or two), resulting in a limited understanding of how different models perform and compare in their refactoring capabilities. Furthermore, the impact of other factors such as reasoning ability, model size or the difficulty of the underlying tasks on refactoring performance remains largely unexplored.

 doi:10.20944/preprints202603.0295.v1

In this work, we perform an extensive study on the refactoring capabilities of smaller open-weight language models (SLMs). Such models are particularly interesting because they are cheaper and easier to run locally, making them more accessible in practice. Our investigation includes 12 models, ranging from 0.5B to 8B parameters, covering general-purpose models [17,18], coding models [1,19], and ones with reasoning capabilities [18]. To the best of our knowledge, our study is the first to provide such a broad analysis of the refactoring capabilities of open SLMs. Using a dataset of 3,453 Python programs from the APPS benchmark [20], we assess how well these models can (i) preserve the original behavior of code and (ii) improve code quality. Behavior preservation is measured by running unit tests, while the extent to which SLMs improve code quality is assessed through metrics reflecting factors like code complexity and compliance with PEP-8. In addition, we explore how model size, reasoning ability, and the difficulty of the underlying tasks affect refactoring quality.

Most of the evaluated models show promising performance in code refactoring, but their ability to maintain program behavior is not always consistent. We observed the following trends in their current performance:

- Larger models generally achieve better performance, while smaller models struggle to preserve program equivalence.
- Task complexity negatively impacts performance, with behavior-preserving modifications decreasing as difficulty increases.
- Reasoning improves refactoring quality for larger models but may hinder smaller ones, indicating that its benefits also depend on model size.
- When equivalence is not preserved, errors are predominantly runtime and logical, whereas syntax errors and timeouts are rare.
- Reasoning further reduces the rate of runtime errors in non-equivalent modifications.
- Significant reduction in PEP-8 violations reveal that refactorings result in cleaner code that adheres to coding standards.
- While improvements in code complexity and maintainability are observed, these are less notable.

2. Related Work

Shirafuji et al. evaluated the refactoring capabilities of GPT-3.5 on 880 Python programs sourced from the AOJ online judge system [12]. The model was prompted with few-shot examples to produce ten refactored versions of each program, after which only functionally correct outputs were retained for analysis. The refactorings were assessed using metrics such as Cyclomatic Complexity, lines of code, character and token counts, as well as distance and similarity metrics. Their results showed that GPT-3.5 generated functionally correct and less complex programs in 95.68% of cases, achieving an average reduction of 17.35% in Cyclomatic Complexity and 25.84% in average line length.

In a study on the code generation capabilities of GPT-4, Poldrack et al. examined its refactoring capabilities [6]. The authors collected 2,000 Python programs from GitHub, selecting at most one program per repository. They retained 274 programs for analysis after several filtering steps. Syntactically invalid programs were removed using the flake8 linter before refactoring was applied. The refactored outputs were then evaluated using a range of metrics, including flake8 error counts, lines of code, number of comments, Cyclomatic Complexity, Maintainability Index, Halstead Difficulty, and Halstead Bugs. The results showed substantial improvements: mean flake8 errors per line decreased from 0.23 to 0.09, mean Cyclomatic Complexity dropped from 3.46 to 3.28, and the mean Maintainability Index increased from 70.29 to 74.09.

Guo et al. evaluated GPT-3.5-Turbo for automated code refinement within a code review setting [8]. The study compared the performance of the model using an existing benchmark tool (CodeReviewer). Five prompt templates of increasing information content were used, ranging from a minimal prompt to a fully detailed scenario with comprehensive requirements. Refactoring quality was assessed by comparing the generated code to canonical solutions using Exact Match, BLEU, and their trimmed variants. The authors also examined the effect of temperature settings. Their results showed that

 doi:10.20944/preprints202603.0295.v1

GPT-3.5-Turbo performed best when given prompts containing base requirements, the associated code review, and scenario descriptions, and that lower temperatures produced more stable and higher-quality outputs.

Caumartin et al. replicated the study of Guo et al. to evaluate whether open-weight SLMs can serve as viable alternatives to proprietary models [7]. Using the same code review dataset, they assessed the refinement capabilities of Llama 2 7B and CodeLlama 7B by comparing their outputs to canonical solutions. The evaluation followed the original setup and relied on text similarity metrics rather than unit tests. Their results showed that CodeLlama 7B achieved performance comparable to GPT-3.5-Turbo, suggesting that using open-weight models can be a viable alternative.

Cordeiro et al. investigated the refactoring capabilities of StarCoder2 15B using 30 open-weight Java projects from the 20-MAD dataset [9]. The study compared refactorings generated by the model against those produced by developers and analyzed the types of refactoring operations applied. Correctness was assessed using automatically generated unit tests, while code quality improvements were measured through Cyclomatic Complexity and code smell reduction. The authors also examined the impact of prompt engineering, finding that one-shot prompting produced higher unit test pass rates and greater reductions in code smells than zero-shot prompting. The results indicate that while developers perform better on complex refactorings, StarCoder2 excels in systematic and repetitive refactoring tasks, with the best outcomes achieved by combining one-shot prompting and multiple generations.

Liu et al. examined the refactoring capabilities of GPT-4 and Gemini-1.0 Pro using a dataset of 180 real-world refactoring opportunities from 20 Java projects [10]. These opportunities required applying standard refactoring techniques such as extract method and rename variable. The models initially identified only a small fraction of the required refactorings. However, supplying more detailed prompts substantially improved the performance of GPT-4, increasing its success rate from 15.6% to 86.7%. Across all tasks, 63.6% of refactoring recommendations generated by GPT-4 were comparable to or better than developer-crafted refactorings. Both models, however, produced unsafe refactorings that altered program behavior, prompting the authors to introduce a detect-and-reapply strategy. This approach uses thoroughly tested refactoring engines to reapply the model-identified transformations to the original code and successfully reproduced 94.3% of the refactorings without introducing bugs.

Midolo et al. evaluated the refactoring capabilities of GPT-4o-mini [21]. The basis of their work is a dataset of 100 Python classes from the ClassEval benchmark [22]. They use unit tests to evaluate the behavioral correctness of refactorings. Code quality is asessed with Pylint, Flake8 and SonarCloud. Furthermore they also perform readability evaluation using a state-of-the-art readability tool [23]. Based on their findings, generally 8 out of 10 refactorings pass the unit tests. GPT-4o-mini generally improved code quality, especially according to syntax-related metrics (Flake8). However the model struggles with semantically more complex refactorings. They also found that readability scores often decrease as a result of refactoring.

3. Method

In order to evaluate the refactoring performance of SLMs, we use correct solutions of 3,453 tasks written in Python. These tasks originate from the APPS benchmark [20]. The SLM is prompted to refactor the correct programs one by one, which are then validated in terms of equivalence and quality. We run unit tests on the modified program to check if the modifications are behavior-preserving (i.e. the two programs are equivalent). We also employ a wide range of metrics, which aim to capture the degree to which the programs have been modified and improved. Twelve open SLMs from four model families are used and evaluated in our work. We include Qwen3 models [18] to investigate how reasoning affects the quality of refactorings. These hybrid models are usable both with and without generating reasoning traces before the answer.

 doi:10.20944/preprints202603.0295.v1

3.1. Dataset

The experiments utilize programs selected from the APPS benchmark, which was originally designed to evaluate code generation capabilities of LLMs. In the APPS benchmark, LLMs are instructed to generate Python solutions based on problem descriptions. The benchmark is composed of a training and a test set, each containing 5000 tasks that range from simple programming exercises to problems that require advanced algorithmic skills. As the tasks are annotated with their corresponding task difficulty, we investigate the impact of how the difficulty of the underlying task influences refactoring quality.

We chose the test set for our investigation because every task in it is accompanied by unit tests, which is not the case for the training set. The test cases are needed to confirm that the refactored programs preserve their original behavior. Furthermore, several tasks lack correct solutions and are therefore not suitable for our evaluation. In total, there are 3,453 tasks in the test set with correct solutions, which form the basis for our experiments. The final dataset used for our experiments contains one randomly selected correct solution per each task.

3.2. Validating the behavior-preserving nature of refactorings

Refactoring needs to preserve behavior while modifying the code. That is, a correct refactoring must result in a program that is functionally equivalent to the original. Although a full formal verification of this equivalence is outside the scope of our paper, we can use test cases to evaluate this aspect. Since every program in our dataset passes its associated set of test cases, we rely on these to validate that the refactored versions also preserve the same functionality. For each model output, we check if the refactored code continues to pass the test cases.

If a refactored program doesn't pass every test case, then they are further categorized by the type of failure. We include four failure categories: syntax error, runtime error, logical error and timeout (15 seconds). This further classification gives us the opportunity to look into the cause of failed refactorings of each model.

3.3. Measuring Code Improvement with Code Metrics

In order to determine the degree to which the programs have been improved as the result of refactoring, we use multiple metrics. These metrics aim to capture code complexity, adherence to formatting guidelines, and the extent to which the refactored programs differ from their original counterparts. For each SLM, we compute metrics for both the original and refactored programs and report their mean values and relative differences ($\frac{refactored-original}{original}$).

In order to measure the difference in terms of complexity, we compute Halstead metrics (Volume, Difficulty, and Effort), as well as Cyclomatic Complexity of both the original and refactored versions of the programs. It is reasonable to expect refactored programs to be less complex compared to the original ones, thus we expect the refactored programs to have lower values for these metrics.

In order to measure if refactorings increase code maintainability, we compute the Maintainability Index score of both the original and refactored versions of the programs. Refactored programs should be more maintainable, therefore we expect the refactored programs to have higher Maintainability Index values.

To measure adherence to formatting guidelines, we evaluate both the original and refactored programs using the *flake8* package. We count the number of warnings and errors in both versions. As refactorings should decrease these, we expect lower values for the refactored programs according to these metrics.

To get a more refined sense of the degree to which the programs change, we compute the similarity between the original and refactored programs. This is measured with CodeBleu [24] as well as Levenstein similarity [25]. We also report the number of lines of code (LOC), logical lines of code (LLOC), source lines of code (SLOC), comments and blanks for both programs in the Appendix. These

 doi:10.20944/preprints202603.0295.v1

support metrics do not factor into the evaluation in the same way as the previously mentioned ones, as their relationship to code quality is less clear.

3.4. Evaluated models and prompt

Our experiments are performed using twelve instruction-tuned[1] open SLMs, belonging to four model families. Specifically, the models used are Qwen2.5-Coder (0.5B, 1.5B, 3B, 7B) [1], Qwen3 (0.6B, 1.7B, 4B, 8B) [18], DeepSeek-Coder (1.3B, 6.7B) [19], and Llama3.2 (1B, 3B) [17]. We choose multiple models from certain families to isolate the effect of model size on refactoring quality: by using models that share identical or similar training methods, we can conclude how their size influences the quality of refactorings. The Qwen3 models are used both with reasoning enabled as well as disabled, facilitating the analysis of how reasoning affects refactoring quality.

The prompt instructs the models to refactor the given Python code without introducing any new libraries or modules. This constraint is critical for the testing environment, as an otherwise correct refactoring could be rejected due to the introduction of a package which is not part of the testing environment. The prompt was formulated as follows:

````
Refactor the following program without introducing any new
libraries or modules.

```python
{code}
```
````

To guide models in generating code, we start their response with the ```` ```python ```` substring, which indicates the start of a Python code block. Due to this constraint, models are encouraged to generate Python code as the first part of their response. This is helpful for two reasons: (1) it ensures that the models have the same context for generating code (only the input code and the instruction to refactor it), allowing a fair comparison of their outputs; and (2) it speeds up the evaluation process by preventing models from generating large amounts of texts before starting to code. Then whenever the ```` ``` ```` substring is generated, we stop the generation. Naturally, Qwen3 models with reasoning turned on are exceptions to this: they are first allowed to generate reasoning tokens, but once the reasoning is complete (marked by the special `</think>` token), they are also forced to proceed to code generation.

As for generation parameters, we set *max_new_tokens* to twice the length of the input code and used sampling with a temperature of (T=0.5) to balance between diversity and quality. We kept all other generation parameters at their default values. An exception to these is the Qwen3 models, for which we used settings recommended by model authors: *enable_thinking=True, do_sample=True, temperature=0.6, top_k=20, top_p=0.95, and min_p=0.0*. Additionally, as reasoning traces can be long, we removed the *max_new_tokens* limit to ensure that they are not cut off prematurely.

4. Results

In this section, we present the results of our experiments. We first report the number and percentage of equivalent refactorings estimated using test cases, which indicate how often models are able to preserve the original program behavior. Second, we report code quality metrics to assess how the refactorings affect program quality.

Table 1 presents an overview of model performance in terms of preserving program behavior, showing the number and percentage of behavior-preserving refactorings. Models are grouped by family and the respective model sizes are also indicated. We visualize the effect of model size in Figure 1.

[1] Although some models originally include "Instruct" in their names, we omit this for brevity. All of the models used in our work are instruction-tuned models.

 doi:10.20944/preprints202603.0295.v1

Table 1. Number and percentage of equivalent modifications when refactoring with each model.

Model	Size	Equivalent modifications
DeepSeek-Coder	1.3B	2118 (61.3%)
	6.7B	2359 (68.3%)
Llama 3.2	1B	583 (16.9%)
	3B	808 (23.4%)
QwenCoder 2.5	0.5B	2278 (66.0%)
	1.5B	1157 (33.5%)
	3B	2274 (65.9%)
	7B	2264 (65.6%)
Qwen3	0.6B	1984 (57.5%)
	0.6B (r)	1089 (31.5%)
	1.7B	2244 (65.0%)
	1.7B (r)	1881 (54.5%)
	4B	2609 (75.6%)
	4B (r)	2838 (82.2%)
	8B	2709 (78.5%)
	8B (r)	2941 (85.2%)

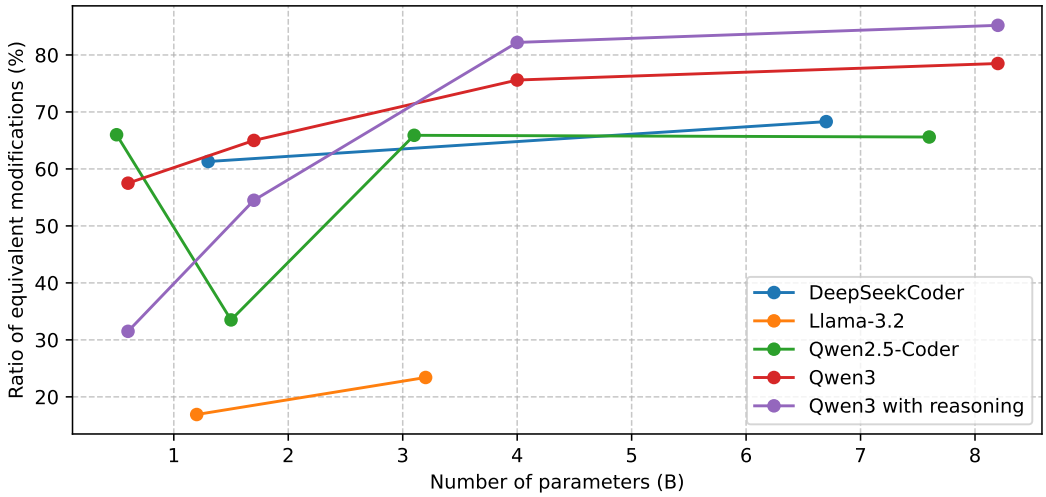

Figure 1. Behavior-preserving refactoring performance by model family and size.

We investigate how the difficulty of the underlying task influences refactoring quality. This can easily be done, as the APPS dataset contains difficulty labels for each task. Difficulty levels include introductory, interview, and competition. The percentage of behavior preserving refactorings across each difficulty level is shown in Table 2.

 doi:10.20944/preprints202603.0295.v1

Table 2. Number of successful (behavior-preserving) refactorings for each model, broken down by difficulty level. The difficulty labels originate from the APPS benchmark.

Model	Size	Introductory (696)	Interview (2474)	Competition (283)
DeepSeek-Coder	1.3B	451 (64.80%)	1516 (61.28%)	151 (53.36%)
	6.7B	521 (74.86%)	1668 (67.42%)	170 (60.07%)
Llama 3.2	1B	182 (26.15%)	371 (15.00%)	30 (10.60%)
	3B	217 (31.18%)	536 (21.67%)	55 (19.43%)
Qwen2.5-Coder	0.5B	460 (66.09%)	1635 (66.09%)	183 (64.66%)
	1.5B	297 (42.67%)	790 (31.93%)	70 (24.73%)
	3B	507 (72.84%)	1609 (65.04%)	158 (55.83%)
	7B	498 (71.55%)	1607 (64.96%)	157 (55.48%)
Qwen3	0.6B	488 (70.11%)	1377 (55.66%)	119 (42.05%)
	0.6B (r)	283 (40.66%)	741 (29.95%)	65 (22.97%)
	1.7B	516 (74.14%)	1575 (63.66%)	153 (54.06%)
	1.7B (r)	458 (65.80%)	1317 (53.23%)	106 (37.46%)
	4B	564 (81.03%)	1859 (75.14%)	186 (65.72%)
	4B (r)	607 (87.21%)	2020 (81.65%)	211 (74.56%)
	8B	593 (85.20%)	1925 (77.81%)	191 (67.49%)
	8B (r)	629 (90.37%)	2093 (84.60%)	219 (77.39%)

We examine the types of failures that occur when refactorings do not preserve behavior. Such cases are indicated by the refactored programs not passing test cases. We report the number of syntax errors, runtime errors, logical errors and timeouts. The distribution of failure types for each model is visualized in Table 3.

Table 3. Distribution of failure types for each model when refactorings do not preserve program behavior. In case of non-equivalent modifications, we categorized the refactorings into syntax errors, runtime errors, logical errors and timeouts.

Model	Size	Syntax	Runtime	Logical	Timeouts
DeepSeek-Coder	1.3B	20 (1.50%)	554 (41.50%)	748 (56.03%)	13 (0.97%)
	6.7B	11 (1.01%)	290 (26.51%)	760 (69.47%)	33 (3.02%)
Llama 3.2	1B	369 (12.86%)	1351 (47.07%)	1126 (39.23%)	24 (0.84%)
	3B	274 (10.36%)	860 (32.51%)	1478 (55.88%)	33 (1.25%)
Qwen2.5-Coder	0.5B	719 (61.19%)	160 (13.62%)	289 (24.60%)	7 (0.60%)
	1.5B	26 (1.13%)	877 (38.20%)	1360 (59.23%)	33 (1.44%)
	3B	12 (1.02%)	462 (39.19%)	678 (57.51%)	27 (2.29%)
	7B	14 (1.18%)	412 (34.59%)	734 (61.63%)	31 (2.60%)
Qwen3	0.6B	18 (1.23%)	624 (42.48%)	797 (54.25%)	30 (2.04%)
	0.6B (r)	23 (0.97%)	1022 (43.23%)	1283 (54.27%)	36 (1.52%)
	1.7B	22 (1.82%)	423 (34.99%)	735 (60.79%)	29 (2.40%)
	1.7B (r)	12 (0.76%)	368 (23.41%)	1160 (73.79%)	32 (2.04%)
	4B	6 (0.71%)	266 (31.52%)	565 (66.94%)	7 (0.83%)
	4B (r)	4 (0.65%)	108 (17.56%)	489 (79.51%)	14 (2.28%)
	8B	2 (0.27%)	182 (24.46%)	544 (73.12%)	16 (2.15%)
	8B (r)	8 (1.56%)	61 (11.91%)	433 (84.57%)	10 (1.95%)

We also characterize the degree to which models improved code quality. We measure this by computing code quality metrics on the original as well as on their refactored counterparts. The used metrics are Cyclomatic Complexity, Maintainability Index, Halstead metrics (Volume, Difficulty, Effort), PEP-8 errors and warnings, CodeBLEU and Levenstein similarity. Table 5 contains the average relative changes in terms of code quality metrics for all refactorings, while Table 4 focuses only on behavior-preserving refactorings. A more detailed analysis with the average metric values are provided in the Appendix.

 doi:10.20944/preprints202603.0295.v1

Table 4. Average relative changes for code quality metrics, with a focus on behavior-preserving refactorings. Code equivalence success rate is also included for reference.

Metric	DeepSeek-Coder		Llama 3.2		Qwen2.5-Coder			
	1.3B	6.7B	1B	3B	0.5B	1.5B	3B	7B
Cyclomatic Complexity	-1.21%	-1.84%	+6.57%	+2.87%	-0.03%	+0.41%	-1.72%	-1.71%
Maintainability Index	-2.13%	-2.66%	+6.00%	+21.09%	-0.61%	+2.90%	+0.47%	+1.46%
Halstead Volume	-0.66%	-0.79%	+6.14%	+3.41%	+0.05%	+0.58%	+0.12%	-1.10%
Halstead Difficulty	-0.20%	+1.17%	+2.84%	-0.36%	+0.01%	-0.96%	+0.48%	+1.54%
Halstead Effort	-0.88%	-0.01%	+6.65%	+3.56%	+0.01%	+0.54%	+0.83%	-0.55%
PEP-8 Errors	-49.53%	-51.75%	-59.97%	-73.41%	-4.88%	-35.34%	-78.30%	-67.43%
PEP-8 Warnings	-56.86%	-70.44%	-52.03%	-68.81%	-3.76%	-64.09%	-80.90%	-79.12%
CodeBLEU Score	0.76	0.68	0.61	0.47	0.98	0.64	0.66	0.60
Levenshtein Similarity	0.90	0.86	0.70	0.49	0.99	0.78	0.85	0.81
Code equivalence	61.3%	68.3%	16.9%	23.4%	66.0%	33.5%	65.9%	65.6%

Metric	Qwen3							
	0.6B	0.6B (r)	1.7B	1.7B (r)	4B	4B (r)	8B	8B (r)
Cyclomatic Complexity	-1.43%	-4.11%	-2.31%	-4.71%	-2.36%	-6.46%	-1.53%	-7.37%
Maintainability Index	+0.02%	-0.22%	+2.75%	-0.40%	+2.64%	+6.26%	+0.84%	+2.30%
Halstead Volume	-0.49%	-1.94%	-0.72%	-4.44%	-0.86%	-7.26%	-1.57%	-10.14%
Halstead Difficulty	+0.32%	-0.50%	+0.62%	-1.22%	+0.64%	-2.67%	+1.48%	-2.67%
Halstead Effort	-0.11%	-1.90%	+0.77%	-4.22%	+0.69%	-7.52%	-0.97%	-10.52%
PEP-8 Errors	-84.11%	-84.99%	-82.50%	-84.41%	-82.99%	-85.18%	-84.69%	-86.46%
PEP-8 Warnings	-62.88%	-79.42%	-77.00%	-80.81%	-74.36%	-74.48%	-81.70%	-78.23%
CodeBLEU Score	0.75	0.64	0.69	0.53	0.63	0.50	0.61	0.49
Levenshtein Similarity	0.91	0.85	0.87	0.78	0.83	0.72	0.82	0.73
Code equivalence	57.5%	31.5%	65.0%	54.5%	75.6%	82.2%	78.5%	85.2%

 doi:10.20944/preprints202603.0295.v1

Table 5. Average relative changes for code quality metrics, including both behavior-preserving and non-preserving refactorings. Code equivalence success rate is also included for reference.

Metric	DeepSeek-Coder		Llama 3.2		Qwen2.5-Coder			
	1.3B	6.7B	1B	3B	0.5B	1.5B	3B	7B
Cyclomatic Complexity	-3.57%	-3.97%	-0.94%	-1.63%	-0.99%	-7.40%	-2.56%	-3.83%
Maintainability Index	-2.22%	-2.56%	+14.64%	+25.28%	-0.58%	+8.32%	+1.34%	+2.85%
Halstead Volume	-3.08%	-3.96%	-4.56%	-2.94%	-1.05%	-11.07%	-0.58%	-4.33%
Halstead Difficulty	-1.74%	-1.76%	-5.45%	-5.03%	-0.95%	-8.71%	-0.63%	+1.15%
Halstead Effort	-4.16%	-9.01%	-7.07%	-8.35%	-1.71%	-19.71%	+0.36%	-3.80%
PEP-8 Errors	-54.19%	-55.34%	-71.18%	-75.75%	+97.48%	-55.46%	-78.47%	-67.90%
PEP-8 Warnings	-73.46%	-66.99%	-77.14%	-74.39%	-20.07%	-72.78%	-81.79%	-82.17%
CodeBLEU Score	0.69	0.65	0.47	0.39	0.90	0.51	0.63	0.57
Levenshtein Similarity	0.86	0.84	0.60	0.45	0.87	0.68	0.83	0.78
Code equivalence	61.3%	68.3%	16.9%	23.4%	66.0%	33.5%	65.9%	65.6%

Metric	Qwen3							
	0.6B	0.6B (r)	1.7B	1.7B (r)	4B	4B (r)	8B	8B (r)
Cyclomatic Complexity	-5.86%	-13.39%	-3.49%	-6.78%	-2.66%	-6.31%	-1.45%	-7.00%
Maintainability Index	+0.85%	+2.83%	+4.88%	+0.23%	+3.31%	+6.50%	+1.10%	+2.33%
Halstead Volume	-6.49%	-15.86%	-2.81%	-7.77%	-1.65%	-8.15%	-1.62%	-9.84%
Halstead Difficulty	-2.48%	-8.34%	-1.69%	-4.48%	+0.18%	-2.73%	+1.75%	-2.72%
Halstead Effort	-10.64%	-21.01%	-6.85%	-11.65%	+1.12%	-8.79%	-0.75%	-10.21%
PEP-8 Errors	-86.29%	-86.75%	-84.75%	-85.46%	-83.87%	-85.65%	-85.75%	-86.61%
PEP-8 Warnings	-81.91%	-85.25%	-75.47%	-80.47%	-79.45%	-73.89%	-80.22%	-78.01%
CodeBLEU Score	0.69	0.53	0.65	0.52	0.62	0.49	0.60	0.48
Levenshtein Similarity	0.87	0.77	0.84	0.76	0.82	0.72	0.81	0.72
Code equivalence	57.5%	31.5%	65.0%	54.5%	75.6%	82.2%	78.5%	85.2%

In order to provide a better insight into the style of the dataset as well as the refactorings generated by the models, we also provide a sample of refactorings in the Appendix.

5. Discussion

Probably the most important aspect of refactoring is to ensure that the behavior of the original program is preserved. This is important because otherwise refactorings could have unintended side effects or could inject bugs into the code. Our experiments show that many of the evaluated models perform poorly in this aspect, as they often fail to preserve external behavior. The worst performing models are the two Llama 3.2 models: they refactored only **16.9%** (1B) and **23.4%** (3B) of the programs without changing their behavior. Furthermore, even the best performing model in terms of behavior preservation (Qwen3 8B with reasoning turned on) fails to maintain equivalence in **14.8%** of the cases. The lack of consistent behavior preservation implies that the evaluated models are not suitable for completely reliable refactoring. In production, this limitation could be mitigated with additional safeguards to ensure correctness, such as more rigorous testing or human oversight.

By including models of varying sizes from the same family, we can observe how model size relates to refactoring quality. In most cases, larger models tend to perform better in terms of preserving program behavior than their smaller counterparts within the same family. These trends are visualized in Figure 1. The Qwen2.5-Coder models are especially interesting, as their four versions include a notable outlier in the 0.5B model: it has a success rate of **66.0%**, which is surprisingly high given its small size. However, this high success rate is due to the model frequently copying the original code without making any changes, thus trivially preserving behavior. This can be seen from the high CodeBLEU and Levenstein similarity scores, as well as the minimal changes in lines of code and other metrics (see Table 4 and Table A1).

 doi:10.20944/preprints202603.0295.v1

Similarly to model size, task complexity appears to have an impact on refactoring performance. In general, models perform best on simpler, introductory-level tasks, with their success rate gradually decreasing as task complexity increases: on average, from **65.29%** at the introductory level to **57.19%** for interview-level and to **49.12%** in case of competition-level tasks. This pattern suggests that as the logical and structural complexity of the task grows, models struggle more to preserve behavior during refactoring.

To better understand why programs fail, we categorized the types of errors that occur when refactorings do not pass their test cases. We counted syntax errors, runtime errors, logical errors, and timeouts (Table 3). In most cases, the models introduced runtime and logical errors during refactoring, while syntax errors and timeouts were relatively rare. Three models stood out as exceptions: Qwen2.5-Coder-0.5B and the two Llama3.2 models produced significantly more syntax errors than the others. The most notable among them is Qwen2.5-Coder-0.5B, which generated 1.58 times more syntax errors than all the other error types together.

Regarding the role of reasoning in refactoring, our results indicate that it has a mixed efffect on refactoring capabilities. The 0.6B and 1.7B Qwen3 models performed worse with reasoning turned on, while the 4B and 8B models performed better. This suggests that smaller models are not capable of taking advantage of reasoning tokens to aid refactoring as well as larger ones. Independent of the model size, we see that the use of reasoning leads to a notable decrease in CodeBLEU as well as Levenshtein similarity scores. This indicates that reasoning makes the models do more fundamental changes to the code. Reasoning also appears to reduce the proportion of runtime errors compared to other types of errors.

In general, the models tend to reduce program complexity when attempting to refactor programs, but only to a limited extent. This can be seen from the decrease in cyclomatic complexity: the average decrease for behavior-preserving refactorings is **1.68%**, while across all modifications it is **4.49%**. These values indicate that the models are capable of simplifying programs by reducing their complexity. However, larger, more significant modifications are less likely to preserve behavior, as reflected by the smaller complexity decrease for behavior-preserving refactorings than for all modifications.

The decrease in code complexity is further characterized by Halstead metrics. For the behavior-preserving modifications, Halstead volume decreases on average by **1.23%**, difficulty stays roughly unchanged (increasing only by **0.03%**), and effort decreases by **0.85%**. When considering all modifications, the average decreases are larger, showing a similar trend as observed in case of cyclomatic complexity: **5.36%** in volume, **2.73%** in difficulty, and **7.64%** in effort. In line with observations on cyclomatic complexity, these reductions indicate that the models tend to simplify the overall computational and cognitive load of the programs, but more significant simplifications are less likely to preserve behavior.

Refactored programs also tend to have higher maintainability than their original counterparts, indicated by an increase in Maintainability Index. This score is **2.54%** higher in case of behavior-preserving refactorings, and **4.32%** higher when considering all modifications. Interestingly, the DeepSeek-Coder models, despite their generally good refactoring capabilities (indicated by other code quality metrics and their percentage of behavior-preserving refactorings), tend to modify programs in a way that reduces maintainability. Conversely, although Llama3.2 models are clearly the worst-performing models, code maintainability increases most significantly in their refactorings (by 6% and 21.09% for the 1B and 3B models for the refactorings that preserved behavior and by 14.64% and 25.28% for all the refactorings).

Improvements in code style are much more significant. For behavior-preserving refactorings, the number of PEP-8 errors decreased by **68.50%**, and warnings by **67.81%**. Considering all modifications, errors decreased by **65.37%**, and warnings by **73.97%**. These reductions suggest that the models produce code that is substantially cleaner and more compliant with style guidelines. A notable outlier is Qwen2.5-Coder-0.5B: in case of behavior-preserving refactorings, it achieved a significantly lower reduction in errors (4.88%) and warnings (3.76%) compared to other models. Furthermore, considering

 doi:10.20944/preprints202603.0295.v1

all modifications on average, this model nearly doubled PEP-8 errors (indicated by an increase of 97.48%).

Overall, the evaluated models have varying refactoring performance, especially in terms of preserving program behavior. A common pattern throughout the models is a consistent and significant decrease in PEP-8 violations, which indicates that the models are highly effective at improving adherence to code style. Furthermore, models tend to reduce code complexity and increase maintainability, however, these improvements are clearly less significant. This indicates that while the evaluated models are useful for code refactoring, they tend to focus more on improving adherence to code style rather than on deep code optimizations and simplifications.

6. Conclusion

In this work, we performed a comprehensive study on the refactoring capabilities of small open-weight language models. We evaluated the ability of these models to preserve behavior and improve code quality using a dataset of 3,453 Python programs. We investigated how model size, the ability to reason before constructing the refactored output, and the difficulty of the underlying task may affect refactoring quality. We also examined the types of errors that were encountered while testing the modified programs.

Our investigation shows that although models generally improve code quality, they are not fully reliable in preserving program behavior. Most of the refactorings the models perform are shallow, primarily focused on PEP-8 compliance, while deep structural changes to improve code complexity and maintainability are less significant. When program behavior is not preserved, it is typically due to logical or runtime errors introduced by the models. While better performance can be achieved by using larger models and enabling reasoning (above 4B), these models are only safe to use in professional environments with human oversight.

As future work, we plan to replicate our experiments with larger models. We also plan to use a dataset with larger programs to gain a more refined understanding of refactoring performance. Furthermore, we also plan to experiment with different prompting techniques, such as including more context about the program to be refactored.

Broader Impact

With this paper we hope to contribute to the ongoing effort to democratize generative AI in software development-related tasks, emphasising correct and high-quality code. By identifying current SLM capabilities in source code refactoring, we hope to help stakeholders select the most appropriate models for code refactoring.

Our environmental impact was relatively small: the experiments consumed roughly 500 GPU-hours on two Nvidia A100 GPUs. Our findings allow others to avoid wasted computation that would occur when using SLMs that have been found to be inefficient in code refactoring.

Author Contributions: Conceptualization, T.M. and B.S.; methodology, T.M. and B.S.; software, T.M., B.S.; investigation, T.M.; writing—original draft preparation, T.M.; writing—review and editing, B.S., T.M., B.P. and T.G.; visualization, T.M. and B.S.; supervision, B.P. and T.G.; project administration, T.G.; funding acquisition, T.G. All authors have read and agreed to the published version of the manuscript.

Funding: Supported by the EKÖP-25 University Excellence Scholarship Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

In this appendix, we provide a more detailed overview of the changes in code quality metrics observed across our experiments. Table A1 and Table A2 are extended versions of Table 4 and Table 5. They extend the previously provided relative differences by including the average metric values for

 doi:10.20944/preprints202603.0295.v1

the original and refactored programs. We note that most metrics can only be calculated for parsable programs. Therefore, pairs of programs where the refactoring is unparsable are not included in the averages. As different models usually produce unparsable refactorings for different inputs, the original average metric values can slightly differ for different models. We also provide further metrics in these tables that are not included in the main text: lines of code, logical lines of code, source lines of code, blank lines, and comments.

 doi:10.20944/preprints202603.0295.v1

Table A1. The average relative changes in code quality metrics in case of behavior-preserving modifications. These also include the average metric values for both the original and refactored programs, as well as some additional metrics: lines of code, logical lines, source lines, number of blank lines, and comments.

Metric	Models													
	DeepSeek-Coder				Llama-3.2				Qwen2.5-Coder					
	1.3B		6.7B		1B		3B		0.5B		1.5B		3B		7B	
	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.
Cyclomatic Complexity	7.44	7.35	7.20	7.07	5.69	6.07	6.20	6.38	8.31	8.31	6.54	6.56	7.29	7.16	7.07	6.95
	-1.21%		-1.84%		+6.57%		+2.87%		-0.03%		+0.41%		-1.72%		-1.71%	
Maintainability Index	62.47	61.14	63.00	61.33	66.56	70.55	65.46	79.27	61.19	60.82	64.89	66.77	62.78	63.08	63.24	64.16
	-2.13%		-2.66%		+6.00%		+21.09%		-0.61%		+2.90%		+0.47%		+1.46%	
Halstead Volume	216.64	215.21	206.16	204.52	151.99	161.33	178.71	184.81	243.56	243.68	186.93	188.02	211.47	211.71	199.26	197.07
	-0.66%		-0.79%		+6.14%		+3.41%		+0.05%		+0.58%		+0.12%		-1.10%	
Halstead Difficulty	4.33	4.33	4.20	4.25	3.55	3.65	3.98	3.96	4.60	4.60	3.92	3.89	4.30	4.32	4.19	4.25
	-0.20%		+1.17%		+2.84%		-0.36%		+0.01%		-0.96%		+0.48%		+1.54%	
Halstead Effort	1526.93	1513.55	1384.65	1384.49	991.81	1057.72	1258.53	1303.35	1739.14	1739.30	1241.41	1248.07	1464.93	1477.08	1356.93	1349.45
	-0.88%		-0.01%		+6.65%		+3.56%		+0.01%		+0.54%		+0.83%		-0.55%	
PEP-8 Errors	27.65	13.95	30.09	14.52	17.30	6.92	20.74	5.52	39.80	37.86	22.74	14.71	32.80	7.12	28.68	9.34
	-49.53%		-51.75%		-59.97%		-73.41%		-4.88%		-35.34%		-78.30%		-67.43%	
PEP-8 Warnings	18.00	7.77	21.91	6.48	7.02	3.37	10.97	3.42	24.20	23.30	12.92	4.64	19.16	3.66	16.67	3.48
	-56.86%		-70.44%		-52.03%		-68.81%		-3.76%		-64.09%		-80.90%		-79.12%	
Lines of Code	24.77	23.58	24.76	22.28	18.92	25.22	21.73	36.00	28.53	28.70	22.64	23.41	24.33	24.70	23.80	23.38
	-4.78%		-10.00%		+33.32%		+65.71%		+0.59%		+3.42%		+1.50%		-1.78%	
Logical Lines of Code	21.15	20.55	20.78	19.66	16.29	18.66	18.17	20.92	24.19	24.22	18.91	19.03	20.77	20.65	20.17	19.37
	-2.84%		-5.41%		+14.59%		+15.13%		+0.10%		+0.63%		-0.56%		-3.93%	
Source Lines of Code	20.59	20.10	20.17	19.25	15.86	17.85	17.88	19.54	23.49	23.52	18.43	18.71	20.20	20.39	19.59	19.09
	-2.40%		-4.55%		+12.57%		+9.29%		+0.17%		+1.50%		+0.94%		-2.54%	
Comments	0.75	0.36	0.84	0.28	0.56	0.79	0.75	3.01	0.93	0.87	0.70	0.99	0.79	0.78	0.78	0.86
	-52.18%		-66.48%		+39.21%		+302.98%		-6.52%		+41.11%		-1.22%		+10.58%	
Blank Lines	3.30	3.06	3.60	2.71	2.52	3.81	2.86	6.32	3.95	4.19	3.39	3.69	3.23	3.50	3.24	3.44
	-7.39%		-24.65%		+51.47%		+120.86%		+5.93%		+8.63%		+8.48%		+6.35%	
CodeBLEU	0.76		0.68		0.61		0.47		0.98		0.64		0.66		0.60	
Levenshtein Similarity	0.90		0.86		0.70		0.49		0.99		0.78		0.85		0.81	
Code equivalence	61.3%		68.3%		16.9%		23.4%		66.0%		33.5%		65.9%		65.6%	

Metric	Qwen3													
	0.6B		0.6B thinking		1.7B		1.7B thinking		4B		4B thinking		8B		8B thinking	
	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.
Cyclomatic Complexity	5.79	5.71	5.56	5.33	6.51	6.36	6.22	5.93	7.37	7.20	7.39	6.92	7.37	7.26	7.61	7.05
	-1.43%		-4.11%		-2.31%		-4.71%		-2.36%		-6.46%		-1.53%		-7.37%	
Maintainability Index	64.91	64.93	65.88	65.74	63.87	65.63	64.41	64.15	62.70	64.36	62.58	66.50	62.73	63.26	62.40	63.83
	+0.02%		-0.22%		+2.75%		-0.40%		+2.64%		+6.26%		+0.84%		+2.30%	
Halstead Volume	149.77	149.04	144.30	141.50	179.56	178.27	170.81	163.22	211.68	209.85	207.27	192.23	211.34	208.02	219.51	197.24
	-0.49%		-1.94%		-0.72%		-4.44%		-0.86%		-7.26%		-1.57%		-10.14%	
Halstead Difficulty	3.65	3.66	3.58	3.56	3.95	3.97	3.87	3.82	4.30	4.33	4.27	4.16	4.38	4.44	4.42	4.30
	+0.32%		-0.50%		+0.62%		-1.22%		+0.64%		-2.67%		+1.48%		-2.67%	
Halstead Effort	875.87	874.86	861.77	845.40	1141.02	1149.75	1078.78	1033.22	1453.68	1463.71	1400.49	1295.16	1557.44	1542.32	1590.45	1423.19
	-0.11%		-1.90%		+0.77%		-4.22%		+0.69%		-7.52%		-0.97%		-10.52%	
PEP-8 Errors	23.55	3.74	21.91	3.29	29.03	5.08	26.40	4.12	32.98	5.61	34.11	5.06	34.25	5.24	35.79	4.84
	-84.11%		-84.99%		-82.50%		-84.41%		-82.99%		-85.18%		-84.69%		-86.46%	
PEP-8 Warnings	10.71	3.98	12.51	2.57	21.55	4.96	21.71	4.17	18.27	4.68	19.77	5.05	23.02	4.21	22.62	4.92
	-62.88%		-79.42%		-77.00%		-80.81%		-74.36%		-74.48%		-81.70%		-78.23%	
Lines of Code	18.59	19.27	17.73	16.86	21.65	22.29	20.82	18.60	24.53	25.35	24.88	24.65	24.78	25.38	25.55	23.83
	+3.65%		-4.94%		+2.93%		-10.66%		+3.34%		-0.92%		+2.44%		-6.73%	
Logical Lines of Code	16.11	15.99	15.46	14.49	18.39	18.30	17.66	16.33	20.99	20.76	21.12	19.84	21.01	20.96	21.69	20.04
	-0.78%		-6.25%		-0.52%		-7.56%		-1.10%		-6.03%		-0.22%		-7.60%	
Source Lines of Code	15.80	15.89	15.17	14.48	17.88	18.10	17.17	16.25	20.34	20.57	20.47	19.65	20.37	20.85	21.02	19.94
	+0.54%		-4.55%		+1.23%		-5.37%		+1.13%		-4.01%		+2.36%		-5.16%	
Comments	0.42	0.32	0.39	0.25	0.62	0.72	0.64	0.25	0.76	0.78	0.77	1.63	0.79	0.71	0.84	0.86
	-23.58%		-36.92%		+15.74%		-60.80%		+2.92%		+111.79%		-9.70%		+2.56%	
Blank Lines	2.30	3.06	2.12	2.14	3.01	3.45	2.91	2.09	3.27	3.97	3.50	3.30	3.49	3.81	3.56	3.03
	+33.27%		+0.61%		+14.55%		-28.18%		+21.39%		-5.66%		+9.09%		-14.96%	
CodeBLEU	0.75		0.64		0.69		0.53		0.63		0.50		0.61		0.49	
Levenshtein Similarity	0.91		0.85		0.87		0.78		0.83		0.72		0.82		0.73	
Code equivalence	57.5%		31.5%		65.0%		54.5%		75.6%		82.2%		78.5%		85.2%	

 doi:10.20944/preprints202603.0295.v1

Table A2. The average relative changes in code quality metrics in case of all modifications. These also include the average metric values for both the original and refactored programs, as well as some additional metrics: lines of code, logical lines, source lines, number of blank lines, and comments.

Metric	Models													
	DeepSeek-Coder				Llama-3.2				Qwen2.5-Coder					
	1.3B		6.7B		1B		3B		0.5B		1.5B		3B		7B	
	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.
Cyclomatic Complexity	8.05	7.76	8.04	7.72	7.88	7.81	8.15	8.02	8.67	8.58	8.03	7.44	8.05	7.84	8.02	7.71
		-3.57%		-3.97%		-0.94%		-1.63%		-0.99%		-7.40%		-2.56%		-3.83%
Maintainability Index	61.75	60.38	61.76	60.18	61.93	71.00	61.53	77.09	60.76	60.41	61.78	66.92	61.77	62.59	61.79	63.55
		-2.22%		-2.56%		+14.64%		+25.28%		-0.58%		+8.32%		+1.34%		+2.85%
Halstead Volume	233.92	226.73	234.50	225.21	229.64	219.16	238.72	231.70	256.46	253.77	232.52	206.78	234.52	233.17	232.58	222.50
		-3.08%		-3.96%		-4.56%		-2.94%		-1.05%		-11.07%		-0.58%		-4.33%
Halstead Difficulty	4.52	4.44	4.59	4.50	4.49	4.25	4.66	4.42	4.72	4.68	4.58	4.18	4.58	4.56	4.51	4.56
		-1.74%		-1.76%		-5.45%		-5.03%		-0.95%		-8.71%		-0.63%		+1.15%
Halstead Effort	1680.40	1610.45	1754.37	1596.31	1635.80	1520.14	1797.37	1647.23	1891.60	1859.17	1731.71	1390.39	1753.52	1759.79	1664.75	1601.56
		-4.16%		-9.01%		-7.07%		-8.35%		-1.71%		-19.71%		+0.36%		-3.80%
PEP-8 Errors	35.00	16.03	34.89	15.58	32.63	9.40	31.87	7.73	17.30	34.17	33.72	15.02	36.63	7.89	34.20	10.98
		-54.19%		-55.34%		-71.18%		-75.75%		+97.48%		-55.46%		-78.47%		-67.90%
PEP-8 Warnings	27.14	7.20	26.25	8.66	21.25	4.86	19.45	4.98	27.86	22.27	21.46	5.84	23.39	4.26	22.68	4.04
		-73.46%		-66.99%		-77.14%		-74.39%		-20.07%		-72.78%		-81.79%		-82.17%
Lines of Code	27.13	24.99	27.12	23.93	26.71	34.57	27.35	43.15	29.86	31.59	27.14	28.01	27.13	27.27	27.09	26.06
		-7.87%		-11.75%		+29.46%		+57.77%		+5.76%		+3.20%		+0.52%		-3.81%
Logical Lines of Code	23.00	21.77	23.00	21.17	22.67	24.14	23.20	25.27	25.25	26.98	23.02	22.00	23.01	22.65	22.97	21.41
		-5.36%		-7.95%		+6.46%		+8.91%		+6.84%		-4.41%		-1.54%		-6.79%
Source Lines of Code	22.32	21.30	22.31	20.73	21.99	22.96	22.52	23.56	24.50	26.26	22.34	21.53	22.33	22.34	22.29	21.08
		-4.56%		-7.11%		+4.40%		+4.65%		+7.17%		-3.61%		+0.07%		-5.44%
Comments	0.89	0.37	0.89	0.28	0.85	1.84	0.88	4.06	1.04	0.90	0.90	1.76	0.90	0.96	0.89	1.09
		-59.16%		-68.23%		+115.83%		+361.16%		-13.74%		+96.56%		+6.54%		+21.43%
Blank Lines	3.76	3.27	3.76	2.88	3.70	5.29	3.79	7.69	4.14	4.32	3.76	4.67	3.76	3.95	3.76	3.95
		-12.96%		-23.38%		+42.95%		+102.73%		+4.36%		+24.19%		+5.04%		+5.27%
CodeBLEU	0.69		0.65		0.47		0.39		0.90		0.51		0.63		0.57	
Levenshtein Similarity	0.86		0.84		0.60		0.45		0.87		0.68		0.83		0.78	
Code equivalence	61.3%		68.3%		16.9%		23.4%		66.0%		33.5%		65.9%		65.6%	

Metric	Qwen3													
	0.6B		0.6B thinking		1.7B		1.7B thinking		4B		4B thinking		8B		8B thinking	
	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.	orig.	ref.
Cyclomatic Complexity	8.03	7.56	8.04	6.96	8.03	7.75	8.03	7.48	8.06	7.85	8.05	7.55	8.05	7.94	8.06	7.49
		-5.86%		-13.39%		-3.49%		-6.78%		-2.66%		-6.31%		-1.45%		-7.00%
Maintainability Index	61.77	62.30	61.78	63.53	61.76	64.77	61.78	61.92	61.75	63.79	61.76	65.77	61.76	62.44	61.76	63.20
		+0.85%		+2.83%		+4.88%		+0.23%		+3.31%		+6.50%		+1.10%		+2.33%
Halstead Volume	233.55	218.39	234.15	197.02	234.33	227.74	233.33	215.21	234.87	230.99	234.87	215.72	234.80	230.99	234.92	211.80
		-6.49%		-15.86%		-2.81%		-7.77%		-1.65%		-8.15%		-1.62%		-9.84%
Halstead Difficulty	4.58	4.47	4.51	4.13	4.59	4.51	4.58	4.38	4.59	4.60	4.59	4.46	4.59	4.67	4.59	4.46
		-2.48%		-8.34%		-1.69%		-4.48%		+0.18%		-2.73%		+1.75%		-2.72%
Halstead Effort	1742.12	1556.74	1681.21	1327.93	1755.21	1634.99	1746.65	1543.21	1756.02	1775.64	1756.92	1602.48	1756.24	1743.02	1757.70	1578.25
		-10.64%		-21.01%		-6.85%		-11.65%		+1.12%		-8.79%		-0.75%		-10.21%
PEP-8 Errors	38.30	5.25	37.25	4.94	37.90	5.78	37.43	5.44	37.83	6.10	38.24	5.49	38.60	5.50	38.64	5.17
		-86.29%		-86.75%		-84.75%		-85.46%		-83.87%		-85.65%		-85.75%		-86.61%
PEP-8 Warnings	23.73	4.29	25.12	3.71	22.25	5.46	24.35	4.75	23.57	4.84	21.16	5.52	23.51	4.65	23.50	5.17
		-81.91%		-85.25%		-75.47%		-80.47%		-79.45%		-73.89%		-80.22%		-78.01%
Lines of Code	27.12	26.66	27.11	23.12	27.11	28.66	27.15	25.34	27.15	28.89	27.14	27.10	27.14	28.02	27.14	25.58
		-1.67%		-14.73%		+5.72%		-6.65%		+6.44%		-0.17%		+3.26%		-5.74%
Logical Lines of Code	23.00	21.83	23.00	19.44	23.00	22.56	23.02	21.21	23.02	22.66	23.02	21.77	23.02	23.05	23.02	21.46
		-5.07%		-15.49%		-1.94%		-7.88%		-1.59%		-5.44%		+0.16%		-6.75%
Source Lines of Code	22.32	21.65	22.32	19.27	22.32	22.30	22.34	21.03	22.34	22.48	22.34	21.59	22.34	22.91	22.34	21.37
		-2.99%		-13.65%		-0.06%		-5.88%		+0.64%		-3.35%		+2.58%		-4.34%
Comments	0.89	0.73	0.89	0.58	0.89	1.98	0.90	1.44	0.90	2.06	0.90	1.84	0.90	0.86	0.89	0.94
		-17.88%		-34.67%		+122.56%		+61.20%		+129.73%		+105.21%		-3.40%		+5.69%
Blank Lines	3.76	4.28	3.75	3.25	3.75	4.37	3.76	2.85	3.76	4.34	3.76	3.59	3.76	4.23	3.76	3.24
		+13.72%		-13.34%		+16.74%		-24.12%		+15.55%		-4.40%		+12.65%		-13.74%
CodeBLEU	0.69		0.53		0.65		0.52		0.62		0.49		0.60		0.48	
Levenshtein Similarity	0.87		0.77		0.84		0.76		0.82		0.72		0.81		0.72	
Code equivalence	57.5%		31.5%		65.0%		54.5%		75.6%		82.2%		78.5%		85.2%	

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 4 March 2026 doi:10.20944/preprints202603.0295.v1

Furthermore, we include a sample of refactorings here in order to provide more insight into the style of programs in the dataset, as well as model performance in code refactoring. Figure A1 shows a behavior preserving refactoring that was generated by the best performing model, namely Qwen3 8B with reasoning enabled. The model simplifies a for loop to a list comprehension. Figure A2 also shows a behavior preserving refactoring, generated by the worst performing model, Llama 3.2 1B. It shows us the tendency of Llama to introduce many new comments while also making little or no actual changes. Figure A3 shows another refactoring generated by the same Llama 3.2 1B model which failed the testing. The model encountered the string "red" in the code snippet, which probably triggered the use of the hardcoded string "green" instead of the appropriate variable call. Code similarity results in case of Qwen2.5-Coder 0.5B indicate that the model tends to keep the programs unchanged. Figure A4 shows a good example of this tendency. Figure A5 shows a refactoring generated by the Qwen3 8B model with reasoning enabled, which failed the testing. Although the model generated a code which is more in line with PEP-8 guidelines, it introduced an error when trying to simplify the for loop. It did not take into account that the loop must iterate all the way through the inputs, and not break once `b == c` is true.

```python
n = int(input())
lst = [int(x) for x in input().split()]
for i in range(len(lst)):
    if lst[i]%2==0:
        lst[i]-=1
lst = [str(x) for x in lst]
print(' '.join(lst))
```

⇒

```python
n = int(input())
lst = [int(x) for x in input().split()]
lst = [x - 1 if x % 2 == 0 else x for x in
    lst]
print(' '.join(map(str, lst)))
```

Figure A1. Behaviour preserving refactoring made by Qwen3 8B with reasoning turned on. The model replaced the while loop and the if statement with a conditional list comprehension.

```python
a, b = list(map(int, input().split()))
res = a - 1
if a <= b:
    res += 1

print(res)
```

⇒

```python
def main():
    # Get two integers from the user
    a, b = map(int, input().split())

    # Calculate the result
    res = a - 1
    if a <= b:
        res += 1

    # Print the result
    print(res)

if __name__ == "__main__":
    main()
```

Figure A2. Behaviour preserving refactoring made by Llama 3.2 1B. The model made virtually no changes to the code, except for adding comments and extracting it into a main function.

```python
a=int(input())
s=input()

print(s if a>=3200 else "red")
```

⇒

```python
def main():
    # Get user input
    a = int(input())
    s = input()

    # Check if the number is greater than or
        equal to 3200
    if a >= 3200:
        # Print the message for successful
            input
        print("green")
    else:
        # Print the message for invalid
            input
        print("red")

if __name__ == "__main__":
    main()
```

Figure A3. Refactoring made by Llama 3.2 1B which failed the testing. The model confidently added some descriptive comments, but failed to understand that it needed to print the value of a variable instead of a hardcoded string.

 doi:10.20944/preprints202603.0295.v1


```
import time
li = list(map(int, input().split()))
n = li.pop(0)
li.sort()
for i in range(20000000):
    s = 0
print(" ".join(map(str, li)))
```

⇒

```
import time
li = list(map(int, input().split()))
n = li.pop(0)
li.sort()
for i in range(20000000):
    s = 0
print(" ".join(map(str, li)))
```

Figure A4. Behavior preserving refactoring made by Qwen2.5-Coder 0.5B. This model tends to keep the programs unchanged, as showcased by this example.

```
def solve():
    n, m = list(map(int, input().split()))

    found = False
    for _ in range(n):
        a, b = list(map(int, input().split()
            ))
        c, d = list(map(int, input().split()
            ))
        if b==c:
            found = True

    if m%2==1:
        return False
    if found:
        return True
    return False


t = int(input())

for _ in range(t):
    if solve():
        print("YES")
    else:
        print("NO")
```

⇒

```
def solve():
    n, m = map(int, input().split())
    found = False
    for _ in range(n):
        a, b = map(int, input().split())
        c, d = map(int, input().split())
        if b == c:
            found = True
            break
    return (m % 2 == 0) and found

t = int(input())

for _ in range(t):
    if solve():
        print("YES")
    else:
        print("NO")
```

Figure A5. Refactoring generated by Qwen3 8B with reasoning turned on which failed the testing. Even the best performing model in our evaluation can make mistakes.

References

1. Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; et al. Qwen2.5-Coder Technical Report, 2024, [arXiv:cs.CL/2409.12186].
2. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. Code Llama: Open Foundation Models for Code, 2024, [arXiv:cs.CL/2308.12950].
3. OpenAI.; Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F.L.; Almeida, D.; Altenschmidt, J.; Altman, S.; et al. GPT-4 Technical Report, 2024, [arXiv:cs.CL/2303.08774].
4. Muennighoff, N.; Liu, Q.; Zebaze, A.; Zheng, Q.; Hui, B.; Zhuo, T.Y.; Singh, S.; Tang, X.; Von Werra, L.; Longpre, S. OctoPack: Instruction Tuning Code Large Language Models. In Proceedings of the International Conference on Learning Representations; Kim, B.; Yue, Y.; Chaudhuri, S.; Fragkiadaki, K.; Khan, M.; Sun, Y., Eds., 2024, Vol. 2024, pp. 7604–7623.
5. *Refactoring: improving the design of existing code*; Addison-Wesley Longman Publishing Co., Inc.: USA, 1999.
6. Poldrack, R.A.; Lu, T.; Beguš, G. AI-assisted coding: Experiments with GPT-4, 2023, [arXiv:cs.AI/2304.13187].
7. Caumartin, G.; Qin, Q.; Chatragadda, S.; Panjrolia, J.; Li, H.; Costa, D.E. Exploring the Potential of Llama Models in Automated Code Refinement: A Replication Study. 03 2025, pp. 681–692. https://doi.org/10.1109/SANER64311.2025.00070.
8. Guo, Q.; Cao, J.; Xie, X.; Liu, S.; Li, X.; Chen, B.; Peng, X. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In Proceedings of the Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, New York, NY, USA, 2024; ICSE '24. https://doi.org/10.1145/3597503.3623306.
9. Cordeiro, J.; Noei, S.; Zou, Y. An Empirical Study on the Code Refactoring Capability of Large Language Models, 2024, [arXiv:cs.SE/2411.02320].
10. Liu, B.; Jiang, Y.; Zhang, Y.; Niu, N.; Li, G.; Liu, H. Exploring the potential of general purpose LLMs in automated software refactoring: an empirical study. *Automated Software Engineering* **2025**, *32*, 26. https://doi.org/10.1007/s10515-025-00500-0.

 doi:10.20944/preprints202603.0295.v1

11. Palit, I.; Sharma, T. Reinforcement Learning vs Supervised Learning: A tug of war to generate refactored code accurately. In Proceedings of the Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering, New York, NY, USA, 2025; EASE '25, p. 429–440. https://doi.org/10.1145/3756681.3756971.
12. Shirafuji, A.; Oda, Y.; Suzuki, J.; Morishita, M.; Watanobe, Y. Refactoring Programs Using Large Language Models with Few-Shot Examples. In Proceedings of the 2023 30th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2023, pp. 151–160. https://doi.org/10.1109/apsec60848.2023.00025.
13. McCabe, T. A Complexity Measure. *IEEE Transactions on Software Engineering* **1976**, *SE-2*, 308–320. https://doi.org/10.1109/TSE.1976.233837.
14. Gergel, B.; Stroulia, E.; Smit, M.; Hoover, H.J. Maintainability and source code conventions: An analysis of open source projects **2011**.
15. Halstead, M.H. *Elements of Software Science (Operating and programming systems series)*; Elsevier Science Inc.: USA, 1977.
16. Van Rossum, G.; Warsaw, B.; Coghlan, N. PEP 8–style guide for python code. *Python. org* **2001**, *1565*, 28.
17. AI, M. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. Blog post on Meta AI, 2024. Aaccessed 20 November 2025.
18. Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. Qwen3 Technical Report, 2025, [arXiv:cs.CL/2505.09388].
19. Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.K.; et al. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence, 2024, [arXiv:cs.SE/2401.14196].
20. Hendrycks, D.; Basart, S.; Kadavath, S.; Mazeika, M.; Arora, A.; Guo, E.; Burns, C.; Puranik, S.; He, H.; Song, D.; et al. Measuring Coding Challenge Competence With APPS. *NeurIPS* **2021**.
21. Midolo, A.; Tramontana, E.; Penta, M.D. From Human to Machine Refactoring: Assessing GPT-4's Impact on Python Class Quality and Readability, 2026, [arXiv:cs.SE/2601.13139].
22. Du, X.; Liu, M.; Wang, K.; Wang, H.; Liu, J.; Chen, Y.; Feng, J.; Sha, C.; Peng, X.; Lou, Y. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation, 2023, [arXiv:cs.CL/2308.01861].
23. Scalabrino, S.; Linares-Vásquez, M.; Oliveto, R.; Poshyvanyk, D. A comprehensive model for code readability. *J. Softw. Evol. Process* **2018**, *30*. https://doi.org/10.1002/smr.1958.
24. Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; Ma, S. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis, 2020, [arXiv:cs.SE/2009.10297].
25. Levenshtein, V.I.; et al. Binary codes capable of correcting deletions, insertions, and reversals. In Proceedings of the Soviet physics doklady. Soviet Union, 1966, Vol. 10, pp. 707–710.