Refactoring Python Code with LLM-Based Multi-Agent Systems: An Empirical Study in ML Software Projects

Alexander Puma Pucho
Institute of Computing, University of Campinas
Campinas, SP, Brazil
a259936@dac.unicamp.br

Alexandre Mello Ferreira
Institute of Computing, University of Campinas
Campinas, SP, Brazil
alexandre.ferreira@unicamp.br

Elder José Reioli Cirilo
Department of Computer Science, Federal University of São João del-Rei
São João del-Rei, MG, Brazil
elder@ufsj.edu.br

Bruno B. P. Cafeo
Institute of Computing, University of Campinas
Campinas, SP, Brazil
cafeo@unicamp.br

ABSTRACT

Refactoring is essential for improving software maintainability, yet it often remains a validation-intensive and developer-guided task—particularly in Python projects shaped by fast-paced experimentation and iterative workflows, as is common in the machine learning (ML) domain. Recent advances in large language models (LLMs) have introduced new possibilities for automating refactoring, but many existing approaches rely on single-model prompting and lack structured coordination or task specialization. This study presents an empirical evaluation of a modular LLM-based multi-agent system (LLM-MAS), orchestrated through the MetaGPT framework, which enables sequential coordination and reproducible communication among specialized agents for static analysis, refactoring strategy planning, and code transformation. The system was applied to 1,719 Python files drawn from open-source ML repositories, and its outputs were compared against both the original and human-refactored versions using eight static metrics related to complexity, modularity, and code size. Results show that the agent consistently produces more compact and modular code, with measurable reductions in function length and structural complexity. However, the absence of a validation agent led to 281 syntactically invalid outputs, reinforcing the importance of incorporating semantic and syntactic verification to ensure transformation correctness and build trust in automated refactoring. These findings highlight the potential of LLM-based multi-agent systems to automate structural code improvements and establish a foundation for future domain-aware refactoring in ML software.

KEYWORDS

Code Refactoring, Large Language Models, Multi-Agent Systems, Software Maintenance Automation, Machine Learning Projects

1 Introduction

Refactoring improves internal code structure without altering external behavior [10, 16], and is essential for long-term software maintainability—particularly in fast-evolving domains like ML, where rapid prototyping, frequent experimentation, and loosely coupled scripts accelerate structural degradation [23, 28]. Despite the availability of tools for manual and semi-automated refactoring [1, 4, 26], fully automated approaches remain constrained, especially in dynamic languages such as Python, where flexibility and iterative workflows often lead to accumulated technical debt (TD) [23, 28].

LLMs have recently shown promise for automating source code edits [2, 8, 14]. However, many LLM-based refactoring tools rely on single-shot prompting and lack structured coordination, semantic validation, or domain adaptation. Emerging multi-agent frameworks like ChatDev [21], SWE-Agent [31], and MANTRA [30] address these gaps through agent collaboration, but are typically tailored for Java or synthetic benchmarks. Their performance on dynamic, real-world Python code remains underexplored.

In this context, machine learning codebases offer a compelling testbed for evaluating automated refactoring strategies: they are often under-tested, highly experimental, and subject to frequent restructuring [28]. Understanding how general-purpose LLMs behave in such settings provides critical insights into both the potential and the limitations of current automation techniques.

This study presents an exploratory empirical evaluation of a modular LLM-based multi-agent system for automated refactoring. The system consists of three role-specialized agents—for static analysis, refactoring strategy planning, and code transformation—coordinated sequentially via the MetaGPT framework [12]. Evaluation is performed on 1,719 real-world Python files extracted from open-source ML repositories, using eight static maintainability and complexity metrics. Outputs are compared against both the original and human-refactored versions to assess the structural impact of the automated system.

Research Questions

- **RQ1**: In which maintainability metrics does automated refactoring improve the original code?
- **RQ2**: How do human and agent-based refactorings differ across maintainability metrics?
- **RQ3**: What refactoring types are most frequently applied by agents and humans, and how does their distribution differ?

2 Related Work

Traditional Refactoring Approaches. Early refactoring research focused on identifying structural issues and recommending localized improvements. Metrics-based methods targeted *Move Method* [29], while graph-based models aided *Extract Class* refactorings [6]. Tools

like RefactoringMiner 3.0 [1] enhanced abstract syntax tree (AST) differencing with semantic matching, and PyRef [4] extended detection to Python. Approaches like WitchDoctor [9] and RMove [7] added real-time recommendations and ML. While effective, these methods rely heavily on static rules or engineered features, limiting their adaptability and capacity to handle heterogeneous codebases.

LLM-Based Techniques for Code Refactoring. Recent work explores LLMs to support refactoring tasks. DePalma et al. [8] and Liu et al. [14] found that ChatGPT can propose useful edits, but suffers from inconsistencies and limited opportunity detection without prompt engineering. EM-Assist [19] improves reliability by combining LLM output with static analysis and IDE integration. Prompting strategies [24] and hybrid designs [33] show promise for improving code quality. Still, challenges persist—such as hallucination, misalignment, and lack of validation. Alomar et al. [2] observed that developers often issue vague prompts, which LLMs may misinterpret. These limitations suggest the need for structured, collaborative approaches to guide and constrain model behavior.

LLM-Based Multi-Agent Systems for Software Engineering Automation. To overcome these limitations, recent research has introduced multi-agent frameworks where specialized LLMs collaborate to emulate structured workflows. Systems like ChatDev [21] and MetaGPT [12] simulate development pipelines via predefined roles and communication protocols, but mainly target synthetic scenarios or code generation. SWE-Agent [31] focuses on bug fixing, while MANTRA [30] and Siddeeq et al. [25] explore refactoring in statically typed languages like Java or Haskell. Other systems like LocalizeAgent [5] and TransAgent [32] address design-level concerns and language translation, respectively, but do not explicitly target maintainability.

These studies collectively underscore the promise of LLM-based agent collaboration in software engineering tasks. However, few explore its applicability to real-world Python codebases — particularly in domains like ML, where dynamic patterns and technical debt present unique challenges for automation.

2.1 System and Experimental Setup

2.2 System Overview

The system under evaluation is a modular Large Language Model-Based Multi-Agent System designed to automate software refactoring by distributing responsibilities across specialized agents. Each agent is instantiated as an independent LLM process and coordinated through the MetaGPT framework [12].

The architecture consists of three sequential agents: the Code Quality Agent, the Refactoring Strategy Agent, and the Code Transformation Agent. These agents collaborate to progressively transform input source code. Figure 1 illustrates the full interaction flow: step ① feeds the input file (`original.py`) to the Code Quality Agent, which performs static analysis and outputs a structured report ②. This report is consumed by the Refactoring Strategy Agent, which generates a refactoring plan ③. The Code Transformation Agent applies these instructions to the input file to produce the final version (`agent_refactored.py`) ④.

Agents exchange information through serialized JSON files to ensure deterministic, inspectable, and reproducible communication. Each agent used the `open-mixtral-8x22b` model, accessed via a publicly available API. This model was selected for its accessibility and straightforward API integration, making it a practical choice for reproducible experimentation within the multi-agent framework. Inference was performed using a temperature of 0.7, following the default setting provided by the model's API, and a maximum context window of 65,536 tokens. All other parameters (e.g., top-p, frequency penalties) were left at their default values. No fine-tuning or prompt engineering was applied beyond task-level role descriptions.

Although the system is domain-agnostic, its evaluation in this study focuses on Python code extracted from ML repositories, where TD and structural degradation are common [23, 28].

2.3 Dataset Description

The evaluation dataset is derived from MLRefScanner [17], a curated collection of refactoring commits across 199 open-source Python ML repositories. From its public replication package,[1] all commits labeled with `has_refactoring=1` were selected, yielding 15,311 candidate commits.

File-level refactoring instances were extracted using:

- `PyRef` [4], to detect method-level refactorings.
- `PyDriller` [27], to retrieve the corresponding pre- and post-commit file snapshots.

After extraction and filtering, a corpus of 12,599 unique Python files was obtained. To accommodate the context length limitations of LLMs, files exceeding 500 lines were discarded. From the resulting pool, a reproducible random sample of 2,000 instances was selected (`random_state=42`).

Each sample includes:

- `original.py`: The version of the file before the refactoring commit.
- `human_refactored.py`: The version of the file after the human-performed refactoring.

The human-refactored version corresponds to the post-commit snapshot in each labeled instance.

2.4 Experiment Pipeline

Each experimental instance comprises three aligned versions of the same Python file: the original version (`original.py`), the version refactored by a human developer (`human_refactored.py`), and the version generated by the system (`agent_refactored.py`). The original serves as input to the LLM-MAS, which sequentially applies analysis, planning, and transformation steps to produce the agent version. The human version corresponds to the post-commit state retrieved from the dataset.

Figure 2 illustrates the complete experimental setup. The pipeline begins with commit selection from MLRefScanner, followed by normalization and filtering. Each valid instance produces the three file variants described above, which are then evaluated using static maintainability and complexity metrics. These metrics are used to compare structural variations across the original, human-refactored, and agent-refactored versions, providing empirical support for the three research questions.

[1] https://github.com/seal-replication-packages/TOSEM2024

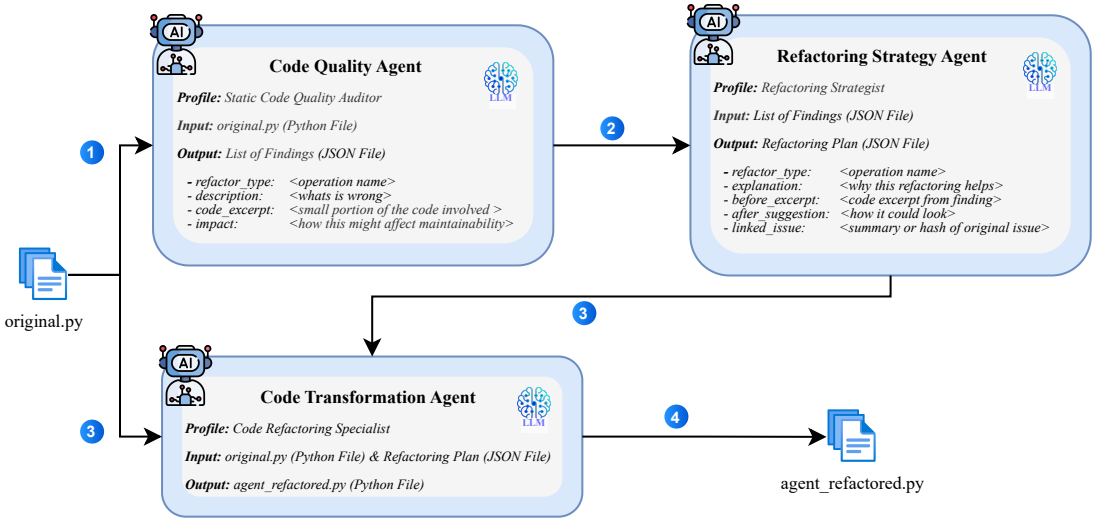

Figure 1: LLM-MAS architecture. Blue numbered circles indicate the data flow: ① Input file; ② Analysis report; ③ Refactoring plan; ④ Transformed code.

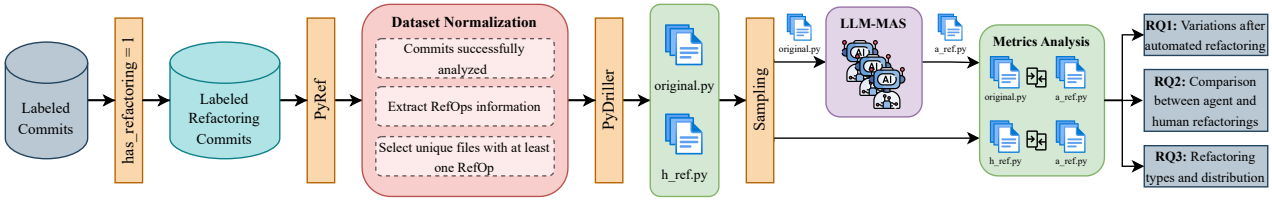

Figure 2: End-to-end experimental pipeline for code refactoring using LLM-MAS: From labeled commits to metrics analysis.

2.5 Evaluation Metrics

To assess the impact on maintainability of automated refactoring, eight static code metrics commonly used in software quality evaluation were selected. These metrics capture aspects of complexity, modularity, documentation, and size. Table 1 summarizes each metric, grouped by the attribute it reflects.

Metrics 1–4 assess structural and cognitive complexity; metrics 5–8 evaluate code size, modularity, and documentation quality. This grouping enables multi-dimensional insight into how refactoring affects maintainability.

All metrics were computed statically using two established tools: *Radon*[2] (for Halstead metrics, lines of code, and cyclomatic complexity) and *Lizard*[3] (for function-level indicators). The analysis did not require code execution, ensuring compatibility with non-runnable or partially complete files.

Each variant (original, human-refactored, and agent-refactored) was evaluated using all metrics. For each metric M, the following deltas were computed to quantify structural change:

$$\Delta_{\text{human}} = M(\texttt{human\_refactored.py}) - M(\texttt{original.py})$$

$$\Delta_{\text{agent}} = M(\texttt{agent\_refactored.py}) - M(\texttt{original.py})$$

$$\Delta_{\text{agent-vs-human}} = M(\texttt{agent\_refactored.py}) - M(\texttt{human\_refactored.py})$$

2.6 Statistical Analysis Plan

The distributions of the computed differences (Δ_{human}, Δ_{agent}, $\Delta_{agent-vs-human}$) were evaluated using the Shapiro-Wilk test, which indicated non-normality ($p < 0.05$). Therefore, all statistical comparisons used non-parametric methods [3, 13].

The Wilcoxon signed-rank test assessed whether the median of paired differences significantly deviated from zero. Comparisons were file-based, using matched pairs to control for project-specific structure. Directional tests determined whether the median was significantly greater than zero (increase, ↑), less than zero (decrease, ↓), or showed no significant change (–), with $\alpha = 0.05$.

Effect sizes were estimated using Cliff's Delta (δ), a robust non-parametric measure. Interpretation followed standard thresholds [22]: negligible ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), moderate ($0.33 \leq |\delta| < 0.474$), and large ($|\delta| \geq 0.474$), reported as –, *, **, and ***, respectively. All calculations were performed in R using the `effsize` package.[4]

[2] https://radon.readthedocs.io/
[3] https://github.com/terryyin/lizard
[4] https://cran.r-project.org/web/packages/effsize/index.html

Table 1: Summary of Code Metrics Used for Evaluation

#	Metric	Definition	Interpretation	Rationale
1	Avg Cyclomatic Complexity (Avg_CC)	Average number of independent execution paths per function	Higher values may indicate more complex logic per function	Logical complexity can affect understandability, maintainability, and testability [15]
2	Halstead Volume (Hal_Vol)	Size of the code in terms of total operators and operands	Larger volume may suggest more verbose or complicated code	Code volume influences cognitive load during comprehension and maintenance [11]
3	Halstead Effort (Hal_Eff)	Estimated mental effort to understand or develop the code	Higher effort implies greater cognitive complexity	Effort correlates with perceived difficulty during code evolution tasks [11]
4	Halstead Difficulty (Hal_Diff)	Logical complexity relative to vocabulary richness	Higher difficulty reflects more intricate logic structures	Logical complexity impacts the ease of program understanding and modification [11]
5	Lines of Code (LoC)	Number of non-comment source lines	Larger codebases may increase maintenance effort	LoC is a traditional proxy for software size and potential change cost [20]
6	Number of Functions (Func_Count)	Number of defined functions in the codebase	Higher modularization may indicate better separation of concerns	Modular systems enhance reusability, readability, and maintainability [10]
7	Comment Density (Comment_Density)	Ratio of comment lines to total lines of code	Higher density suggests more documentation per line of code	Adequate commenting improves code comprehension and maintainability [18]
8	Avg Function Length (Avg_Fun_Len)	Average number of lines per function	Longer functions may imply lower modularity or higher complexity	Smaller functions are associated with better readability and maintainability [10]

3 Results and Discussion

From the 2,000 curated samples described in Section 2.3, a total of 1,719 files were successfully processed and analyzed using the static maintainability metrics defined in Section 2.5. The remaining 281 samples were excluded due to syntactic errors in the LLM-generated code, identified via Python's AST parser. These errors, likely resulting from ill-formed generations, were filtered to preserve the integrity of metric computations.

To contextualize the comparisons, the analysis begins by quantifying the baseline variations induced by human refactorings. Subsequent subsections address RQ1–RQ3 individually.

3.1 Baseline Variations by Human Refactorings

Metric variations resulting from human refactorings were assessed by comparing each `human_refactored.py` file to its corresponding `original.py` version. Table 2 reports the direction, average delta, and effect size for each metric.

Table 2: Direction, Mean Variation, and Effect Size for Human Refactorings

Metric	Δ_{Human}		
	Direction	Mean	δ
Avg_CC	↑	+0.05	–
Hal_Vol	↑	+15.43	*
Hal_Eff	↑	+116.9	*
Hal_Diff	↑	+0.11	–
LoC	↑	+10.04	**
Comment_Density	↓	−0.00	–
Func_Count	↑	+0.43	*
Avg_Func_Len	↑	+0.41	*

Change directions (Wilcoxon signed-rank test): ↑ increase, ↓ decrease, (–) no significant difference. Effect sizes (Cliff's Delta δ): small (*), moderate (**), large (***), negligible (–).

Most maintainability metrics increased after human refactorings. Code size indicators such as `LoC` and `Func_Count` exhibited upward trends, with a moderate effect for `LoC` (**) and small effects for `Func_Count` (*) and `Avg_Func_Len` (*). Complexity-related metrics also showed slight increases—particularly `Hal_Vol` and `Hal_Eff`—with small effect sizes. No statistically significant changes were observed in `Avg_CC`, `Hal_Diff`, or `Comment_Density`, suggesting that these aspects remained relatively stable across refactorings. These results indicate that human developers tend to restructure and modularize code, increasing granularity and length, but do not consistently reduce internal complexity or enhance documentation.

3.2 RQ1: Variations after Automated Refactoring

Paired comparisons between `agent_refactored.py` and `original.py` were used to evaluate whether automated refactorings yield measurable changes. Table 3 summarizes the observed deltas.

Table 3: Direction, Mean Variation, and Effect Size for Agent Refactorings

Metric	Δ_{Agent}		
	Direction	Mean	δ
Avg_CC	↓	-0.26	**
Hal_Vol	-	-8.27	–
Hal_Eff	↓	-76.72	–
Hal_Diff	↓	-0.06	–
LoC	↓	-11.60	*
Comment_Density	↓	-0.03	**
Func_Count	↑	+0.66	**
Avg_Func_Len	↓	-2.49	***

Change directions (Wilcoxon signed-rank test): ↑ increase, ↓ decrease, (–) no significant difference. Effect sizes (Cliff's Delta δ): small (*), moderate (**), large (***), negligible (–).

Most maintainability metrics showed measurable reductions after agent-based refactoring. Notable decreases were observed in `Avg_CC`, `LoC`, `Comment_Density`, and `Avg_Func_Len`, with effect sizes ranging from small to large (* to ***). These results suggest that the automated system is capable of simplifying control flow, shortening functions, and compacting source code.

The agent also increased the number of functions (`Func_Count`, **), possibly indicating a modularization tendency. In contrast, no significant changes were found for `Hal_Vol`, `Hal_Eff`, or `Hal_Diff`, all of which showed negligible effects (–). This suggests that while

structural complexity may be reduced, deeper logical transformations (e.g., effort or difficulty) remain largely unaffected.

These changes align with maintainability principles, as reducing function length, control flow complexity, and code volume generally improves readability, testability, and modular evolution [10].

3.3 RQ2: Comparison between Agent and Human Refactorings

Refactorings produced by the agent were compared directly against those performed by human developers. Table 4 reports average metric differences and effect sizes.

Table 4: Direction, Mean Variation, and Effect Size: Agent vs. Human Refactorings

Metric	$\Delta_{\text{Agent vs Human}}$		
	Direction	Mean	δ
Avg_CC	↓	-0.30	**
Hal_Vol	↓	-23.70	*
Hal_Eff	↓	-193.62	*
Hal_Diff	↓	-0.17	*
LoC	↓	-21.63	***
Comment_Density	↓	-0.03	*
Func_Count	↑	+0.23	*
Avg_Func_Len	↓	-2.90	***

Change directions (Wilcoxon signed-rank test): ↑ increase, ↓ decrease, (–) no significant difference. Effect sizes (Cliff's Delta δ): small (*), moderate (**), large (***), negligible (–).

Compared to human refactorings, the agent produced systematically lower values in most maintainability metrics. Large reductions were observed in `LoC` and `Avg_Func_Len`, with large effect sizes (***), and moderate to small reductions in `Avg_CC`, `Hal_Vol`, `Hal_Eff`, and `Hal_Diff` (* to **). These trends suggest that the agent applies more aggressive structural simplification strategies.

The number of functions increased slightly in agent outputs (`Func_Count`, *), possibly reflecting a modularization behavior not consistently observed in human changes. Comment density was also marginally reduced by the agent (`Comment_Density`, *), suggesting a potential loss of explanatory content compared to the human-refactored version.

Overall, these results indicate that while both humans and agents restructure code, the agent tends to produce more compact and fragmented transformations, reducing complexity and size with greater intensity across several metrics.

3.4 RQ3: Refactoring Types and Distribution

This research question examines which types of refactorings are most frequently applied by the agent and by human developers, and how these refactorings are distributed across files. Table 5 summarizes the frequency of each refactoring category, extracted using PyRef [4].

Overall, the agent applied a substantially higher number of refactorings than humans (12,647 vs. 2,335). While both actors employed a variety of transformation types, agent outputs were dominated by `rename_method` and `extract_method`, suggesting a strong focus on syntactic modularization. In contrast, human developers more frequently applied semantic-preserving changes such as `add_parameter` and `remove_parameter`, indicative of localized behavioral adjustments.

Table 5: Distribution of Refactoring Types (Human vs. Agent)

Refactoring Type	Human Count	Agent Count
Rename Method	541	5132
Extract Method	120	2175
Add Parameter	844	1279
Change/Rename Parameter	259	1126
Change Return Type	100	1083
Rename Class	150	1075
Inline Method	13	628
Remove Parameter	308	138
Other	0	11
Total	2335	12647

Distribution patterns are further illustrated in Figure 3. Most human-refactored files include 1–3 changes, while agent outputs often apply 5 or more, occasionally exceeding 20. This indicates that the agent performs more extensive transformations per file, with a tendency toward applying multiple edits in structurally related locations.

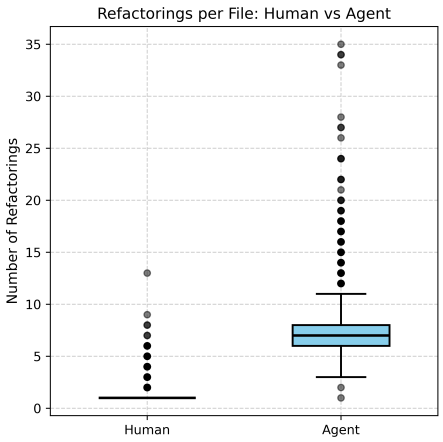

Figure 3: Refactorings per file: Human vs. Agent

While this higher volume may reflect systematic modularization strategies, it also raises the possibility of over-refactoring. Some transformations may be redundant or stylistic rather than impactful, potentially introducing fragmentation or reducing code clarity when performed without semantic awareness. Identifying the boundary between beneficial restructuring and unnecessary edits remains an open challenge. Future work should explore thresholds for meaningful refactoring density and incorporate semantic validation layers to constrain excessive transformation.

Listing 1 shows a representative example. Here, the agent replicates human refactorings and adds changes such as renaming and optional parameterization. While not universally representative, this illustrates the agent's tendency to apply transformations with

```
......
def mean_pairwise_similarity(collection,
       metric=sim, meanfunc=hmean, symmetric=False):
def mean_pairwise_similarity(collection,
       metric=sim, mean_func=hmean, symmetric=False):
def calculate_mean_pairwise_similarity(collection,
       similarity_metric=sim, mean_function=hmean,
    symmetric=False, directionality=None):

"""Calculate the mean pairwise similarity of a
    collection of strings.
......
```

Listing 1: Agent edits outperform human refactorings.
Original, Human, Agent (Human-like), Agent (Outperforms)

broader syntactic consistency and greater intensity. This behavior may reflect a bias toward syntactic patterns, which are more accessible to LLMs than deep behavioral semantics.

These findings reveal a divergence in refactoring strategies: both agents and humans focus on naming and method-level edits, but the agent consistently performs more frequent and wider structural changes, presenting both opportunities and risks for large-scale automation.

4 Threats to Validity

Internal validity. Each `human_refactored.py` file corresponds to a commit labeled as refactoring; however, not all changes within the file are necessarily related to refactoring operations, potentially introducing noise into metric comparisons. On the other hand, `agent_refactored.py` includes only transformations explicitly generated by the system, potentially amplifying the perceived effectiveness of automated refactoring. The system lacks semantic validation to ensure behavioral preservation. All evaluations are limited to syntactic correctness, without verifying whether functional behavior remains intact. A more robust approach would involve executing existing or generated test cases. Another limitation is the exclusion of 281 files due to syntax errors introduced by the agent. Omitting these may bias the evaluation toward successful outputs. Manual or semi-automated analysis of these failures could help identify weaknesses in the refactoring process.

External validity. The evaluation was conducted exclusively on Python codebases, limiting generalizability to other languages. Python's dynamic semantics and flexible syntax differ substantially from statically typed or compiled languages, where refactoring practices and code structure may behave differently. All selected repositories belong to the ML domain, characterized by rapid prototyping and experimental workflows, which may differ from domains like systems programming or embedded software. The dataset was also filtered to exclude files exceeding 500 lines, possibly biasing results toward simpler code and limiting applicability to larger modules.

Construct validity. This study employs static code metrics as proxies for maintainability and complexity. While widely adopted in empirical software engineering, such metrics mostly capture structural aspects and may not reflect semantic clarity, behavioral preservation, or developer intent. The comparison between human and agent-generated refactorings is conducted at the file level, without verifying whether both targeted the same regions or applied equivalent changes. This misalignment weakens claims of equivalence and hampers interpretability. Incorporating semantic comparison or change-alignment mechanisms could enhance future analyses. Refactoring types were identified using PyRef, an automated heuristic tool, which may misclassify complex or non-local changes, affecting the validity of RQ3 findings. Finally, relying on a single pretrained model (`open-mixtral-8x22b`) across all agents may introduce stylistic biases that influence refactoring patterns and metric interpretation.

5 Conclusion

This study evaluated a LLM-MAS for automated refactoring of Python code from machine learning repositories. Across 1,719 files, the system yielded measurable structural improvements—notably in function length, cyclomatic complexity, and code size—compared to both original and human-refactored versions.

The agent consistently applied transformations that promoted modularization and control flow simplification. While human developers tended to make more behavior-focused adjustments, such as parameter changes, the agent primarily performed syntactic restructurings like renaming and method extraction.

The evaluation also revealed key limitations. The absence of semantic validation means behavior preservation was not ensured, and excluding 281 syntactically invalid outputs may have biased results. Additionally, comparisons were made at the file level without confirming that human and agent changes addressed the same regions, limiting interpretability.

These findings highlight the potential of LLM-MAS refactoring, while pointing to essential directions for future work: (i) integrate automated behavioral testing to confirm functional equivalence, (ii) analyze unusable outputs to identify failure modes, (iii) apply semantic-matching techniques to align refactorings, and (iv) tailor strategies to the patterns and workflows of ML codebases. These steps will strengthen confidence in LLM-based refactoring and extend its applicability beyond the current scope.

ARTIFACT AVAILABILITY

The authors declare that the research artifacts supporting the findings of this study are accessible at https://doi.org/10.5281/zenodo.16988695 . It includes the full implementation of the proposed LLM-MAS, all experimental data, refactored code instances, evaluation scripts, and configuration files. The artifact is structured to support inspection, replication, and reuse.

ACKNOWLEDGMENTS

This work was supported by FAEPEX under Grants 3404/23 and 2382/24, and by the R&D projects with Shell Brasil Petróleo Ltda, registered as ANP number 21373-6 and 24282-6, both of them funded by Shell Brazil Technology.

REFERENCES

[1] Pouria Alikhanifard and Nikolaos Tsantalis. 2025. A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–63.

[2] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2024. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 202–206.

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*. 1–10.

[4] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. Pyref: Refactoring detection in python projects. In *2021 IEEE 21st international working conference on source code analysis and manipulation (SCAM)*. IEEE, 136–141.

[5] Fraol Batole, David OBrien, Tien N Nguyen, Robert Dyer, and Hridesh Rajan. 2025. An LLM-Based Agent-Oriented Approach for Automated Code Design Issue Localization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 637–637.

[6] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. 2011. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software* 84, 3 (2011), 397–414.

[7] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. 2022. Rmove: Recommending move method refactoring opportunities using structural and semantic representations of code. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 281–292.

[8] Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. 2024. Exploring ChatGPT's code refactoring capabilities: An empirical study. *Expert Systems with Applications* 249 (2024), 123602.

[9] Stephen R Foster, William G Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 222–232.

[10] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[11] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[12] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).

[13] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust statistical methods for empirical software engineering. *Empirical Software Engineering* 22 (2017), 579–630.

[14] 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, 1 (2025), 26.

[15] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[16] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[17] Shayan Noei, Heng Li, and Ying Zou. 2025. Detecting Refactoring Commits in Machine Learning Python Projects: A Machine Learning-Based Approach. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–25.

[18] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software* 128 (2017), 164–197.

[19] 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 *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 275–287.

[20] Roger S Pressman. 2005. Software Engineering: a practitioner's approach. *Pressman and Associates* (2005).

[21] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).

[22] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*, Vol. 177.

[23] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015).

[24] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 151–160.

[25] Shahbaz Siddeeq, Zeeshan Rasheed, Malik Abdul Sami, Mahade Hasan, Muhammad Waseem, Jussi Rasku, Mika Saari, Kai-Kristian Kemell, and Pekka Abrahamsson. 2025. Distributed Approach to Haskell Based Applications Refactoring with LLMs Based Multi-Agent Systems. *arXiv preprint arXiv:2502.07928* (2025).

[26] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2786–2802.

[27] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 908–911.

[28] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 238–250. https://doi.org/10.1109/ICSE43902.2021.00033

[29] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[30] Yisen Xu, Feng Lin, Jinqiu Yang, Nikolaos Tsantalis, et al. 2025. MANTRA: Enhancing Automated Method-Level Refactoring with Contextual RAG and Multi-Agent LLM Collaboration. *arXiv preprint arXiv:2503.14340* (2025).

[31] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[32] Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. 2024. Transagent: An llm-based multi-agent system for code translation. *arXiv preprint arXiv:2409.19894* (2024).

[33] Zejun Zhang, Zhenchang Xing, Xiaoxue Ren, Qinghua Lu, and Xiwei Xu. 2024. 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.