PeerJ Computer Science

RefactorGPT: a ChatGPT-based multi-agent framework for automated code refactoring

Muhammed Abdulhamid Karabiyik

Department of Computer Technology, Bor Vocational School, Nigde Omer Halisdemir University, Nigde, Turkey

ABSTRACT

The rise of large language models has redefined what is computationally possible in code generation, yet their potential in systematic software refactoring remains largely untapped. This article introduces RefactorGPT, a ChatGPT-augmented sequential multi-agent framework that transforms refactoring from a monolithic, opaque process into a modular, explainable, and scalable workflow. The system orchestrates four specialized agents, Analyzer, Refactor, Refine, and Fixer, to sequentially analyse source code, apply structural refactoring, enhance code quality, and recover from potential generation errors. Unlike conventional rule-based tools or one-shot large language model (LLM) prompts, RefactorGPT leverages ChatGPT iteratively across clearly defined responsibilities, enabling controlled refactoring with functional guarantees. To evaluate the framework's effectiveness, we constructed a curated dataset encompassing nine classical refactoring techniques across three complexity levels. RefactorGPT demonstrated consistent improvements in code modularity, readability, and structural decomposition, while maintaining computational efficiency. Notably, the system achieved full execution correctness through autonomous error recovery, showcasing robustness in practical scenarios. This study contributes a reusable blueprint for LLM-integrated refactoring systems and presents a novel application of ChatGPT not merely as a code generator, but as a cooperative agent in intelligent software refactoring. The findings reveal a path forward for embedding language models into real-world developer workflows, not as assistants, but as collaborators in code evolution.

Submitted 4 June 2025
Accepted 9 September 2025
Published 14 October 2025

Corresponding author
Muhammed Abdulhamid Karabiyik, ma.karabiyik@gmail.com

Academic editor
Armin Mikler

Additional Information and Declarations can be found on page 21

DOI **10.7717/peerj-cs.3257**

Subjects Agents and Multi-Agent Systems, Artificial Intelligence, Natural Language and Speech, Text Mining, Sentiment Analysis
Keywords ChatGPT, Code refactoring, Multi-agent systems, Software automation

INTRODUCTION

Maintaining and improving code quality remains a central concern in modern software engineering, especially as software systems evolve and scale. Among the various techniques used to manage this complexity, code refactoring stands out as a critical practice aimed at improving readability, modularity, and long-term maintainability without altering the external behavior of the software (*Hemel et al., 2010*; *Buse & Weimer, 2010*; *Prinz, 2022*). Despite its well-established benefits, the act of refactoring remains a labor intensive and expertise-dependent process, often requiring in-depth knowledge of software design principles, refactoring strategies, and domain-specific code semantics

(*Bekefi, Szabados & Kovacs, 2019*). This has spurred interest in developing intelligent tools that can automate refactoring tasks without compromising functionality.

In recent years, large language models (LLMs) such as ChatGPT have demonstrated remarkable proficiency in code-related tasks, ranging from generation and summarization to translation and bug fixing. These models leverage vast pretraining corpora and attention-based architectures to produce contextually coherent and syntactically valid outputs (*Liu, 2025*). Their ability to understand code and generate refactoring based on natural language instructions has positioned them as promising candidates for code automation tasks. However, most current applications rely on single-step, monolithic prompting strategies providing a block of code and a refactoring instruction in one go which limits the transparency, control, and correctness guarantees essential in multi-stage processes like refactoring (*Hou, Tang & Wang, 2025*). Such approaches often fall short in scenarios requiring reasoning, iterative refinement, or structured validation.

A growing body of research has explored the potential of LLMs in automating software refactoring. Recent empirical studies show that LLMs can recognize and apply classical refactoring patterns, often matching or exceeding rule-based systems in flexibility and expressiveness. For example, *Shirafuji et al. (2023)* demonstrated that LLMs can perform few-shot refactoring across multiple paradigms with promising results, though their reliability varies depending on prompt structure and code complexity. *Zhang et al. (2024)* proposed a hybrid model that combines LLMs with expert-curated Pythonic idioms, achieving improved stylistic conformity but still depending heavily on human-crafted rules.

Meanwhile, *Wei, Xia & Zhang (2023)* showed that fusing LLMs with traditional completion engines improves success rates in repair tasks, though not specifically tailored for structured refactoring workflows. Complementing these findings, *Liu et al. (2024a)* conducted a broad evaluation of LLMs in refactoring tasks and observed that while LLMs often produce semantically plausible results, they lack mechanisms for stepwise validation and correction. Beyond single-agent designs, more recent work has introduced multi-agent approaches, such as AgentCoder and hierarchical code generation systems, which incorporate iterative feedback loops and task decomposition strategies to improve code quality and correctness over time (*Liu et al., 2024a*; *Akilesh et al., 2025*).

Collectively, these studies highlight both the promise and the current limitations of LLM-driven refactoring. While LLMs excel at flexible code generation, the lack of modular control, validation steps, and structured refinement still present challenges for production-grade applications.

To address these limitations, this study aims to design and evaluate a multi technique, multi language refactoring framework that leverages the flexibility of LLMs while ensuring structured, reproducible, and verifiable transformations. The proposed approach supports nine classical refactoring techniques across Python and Java, enabling broader generalizability compared to prior work that often focuses on a single technique or language. By combining computational and structural evaluation metrics including execution time, memory usage, lines of code (LOC), function count, token count, complexity, and output equivalence checks, the framework provides a balanced assessment

of both performance and maintainability improvements. Furthermore, the use of a systematically constructed, publicly available dataset facilitates transparent benchmarking and fosters future research on LLM based refactoring systems.

Despite increasing interest in automating refactoring tasks, conventional tools often fall short in adapting to real-world coding practices. Rule-based systems are limited by their reliance on predefined refactoring templates, making them rigid and narrow in scope. While effective for well-defined cases, these tools struggle with semantic nuances, nested logic, or unconventional code structures that deviate from their built-in patterns (*Rochimah, Arifiani & Insanittaqwa, 2015*). As a result, developers must often manually intervene when the refactoring context falls outside the bounds of these heuristics.

To overcome the limitations of monolithic prompts and static rule-based tools, this study presents RefactorGPT a modular, ChatGPT-powered refactoring framework based on a multi-agent architecture. It breaks the process into four specialized stages, handled by AnalyzerAgent, RefactorAgent, RefineAgent, and FixerAgent, enabling interpretable and controllable workflows.

Each agent focuses on a specific task such as technique selection, transformation, optimization, or error recovery, ensuring functional reliability and semantic consistency. This multi-stage interaction supports iterative refinement and validation, showcasing ChatGPT not just as a code generator, but as a reasoning engine in structured software workflows. RefactorGPT exemplifies how ChatGPT can act as a collaborative agent for scalable, context-aware code modernization.

This study contributes to the emerging field of ChatGPT-driven software engineering by presenting one of the first modular frameworks for automated code refactoring grounded entirely in large language model interactions. Unlike prior approaches that treat refactoring as a one-off refactoring or an auxiliary feature of code generation, RefactorGPT repositions it as a structured, agent-driven workflow designed to enhance readability, maintainability, and execution reliability. Through empirical evaluation on nine classical refactoring techniques and multi-level complexity examples, the framework demonstrates how a ChatGPT centered architecture can yield consistent improvements in both structural and semantic dimensions of source code.

In doing so, the study provides a reusable blueprint for how ChatGPT can be operationalized as a multi-stage reasoning engine moving beyond prompt-response paradigms into coordinated, feedback-aware code refactoring pipelines. The system's extensible design, interpretable outputs, and runtime validation capabilities make it a promising foundation for future research and real-world deployment. By bridging the flexibility of LLMs with the rigor of modular software processes, RefactorGPT exemplifies a new class of intelligent software assistants designed to enable practical, context-aware integration of ChatGPT into software engineering workflows.

MATERIALS AND METHODS

The technical design of the proposed system is organized around a modular agent-based architecture that enables structured decomposition of the refactoring process into dedicated components. Each agent Analyzer, Refactor, Refine, and Fixer is encapsulated in

its own Python class, ensuring separation of concerns, modularity, and testability (*De Smedt & Daelemans, 2012*). The system leverages OpenAI's GPT-4 API for all language model interactions, while execution-level isolation is achieved *via* temporary subprocess environments to ensure accurate timing and memory profiling (*Sanderson, 2023*). Execution time is measured using Python's standard time library, and peak memory usage is captured with the psutil library. In addition, a lightweight web interface built with Flask allows users to submit Python or Java code, observe step-by-step transformations, and monitor runtime performance. This design ensures both reproducibility for research settings and practical usability for interactive applications. The remainder of this section details the implementation environment and the specific responsibilities of each agent in the RefactorGPT pipeline.

System overview

RefactorGPT is designed as a modular, multi-agent system that automates Python and Java code refactoring through sequential interactions with the ChatGPT language model. The framework consists of four primary agents: AnalyzerAgent, RefactorAgent, RefineAgent, and FixerAgent. Each agent is responsible for a distinct task in the refactoring pipeline and communicates with the model using specialized prompts tailored to that task. This layered design allows for controlled execution, transparent evaluation, and stepwise debugging of the refactoring process.

In this context, each component is conceptualized as an LLM-based agent a task-oriented, self-contained module that leverages a large language model to fulfill a specific role within the refactoring pipeline. Unlike single pass prompting strategies, these agents follow a modular design principle, where responsibilities such as analysis, refactoring, and validation are decoupled into independent, reusable units. This architecture supports greater consistency in outputs, facilitates targeted improvements for each stage, and enables the system to scale or adapt by extending or modifying individual agents without disrupting the overall workflow.

The process begins with the AnalyzerAgent, which identifies the most appropriate refactoring technique based on the input code. This decision is passed to the RefactorAgent, which applies the refactoring according to the selected technique while preserving the original functionality. The resulting code is then enhanced by the RefineAgent, which improves its readability, modularity, and stylistic consistency. Finally, the FixerAgent checks the syntactic and runtime validity of the code and attempts to automatically correct any errors encountered. This agent ensures the robustness of the system and helps maintain code functionality even when the model output is imperfect. The overall data flow between agents is illustrated in Fig. 1.

Figure 2 presents a step-by-step transformation of a simple Java code snippet as it passes sequentially through the AnalyzerAgent, RefactorAgent, and RefineAgent stages of the RefactorGPT framework. The original code, shown in the Input panel, is first analyzed by the AnalyzerAgent, which detects applicable refactoring techniques and identifies the programming language. In this example, Extract Method and Inline Temp are recommended. The RefactorAgent then restructures the code accordingly, extracting the

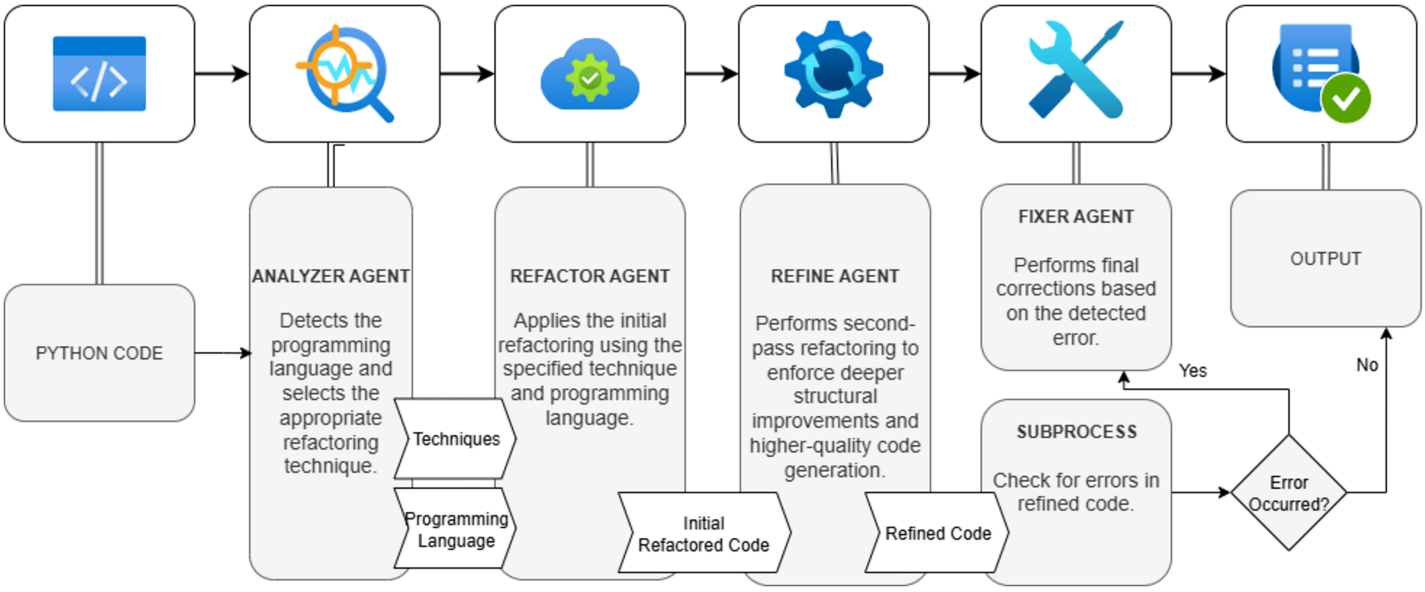

Figure 1 **High-level architecture and data flow of the RefactorGPT multi-agent pipeline.** Full-size DOI: 10.7717/peerj-cs.3257/fig-1

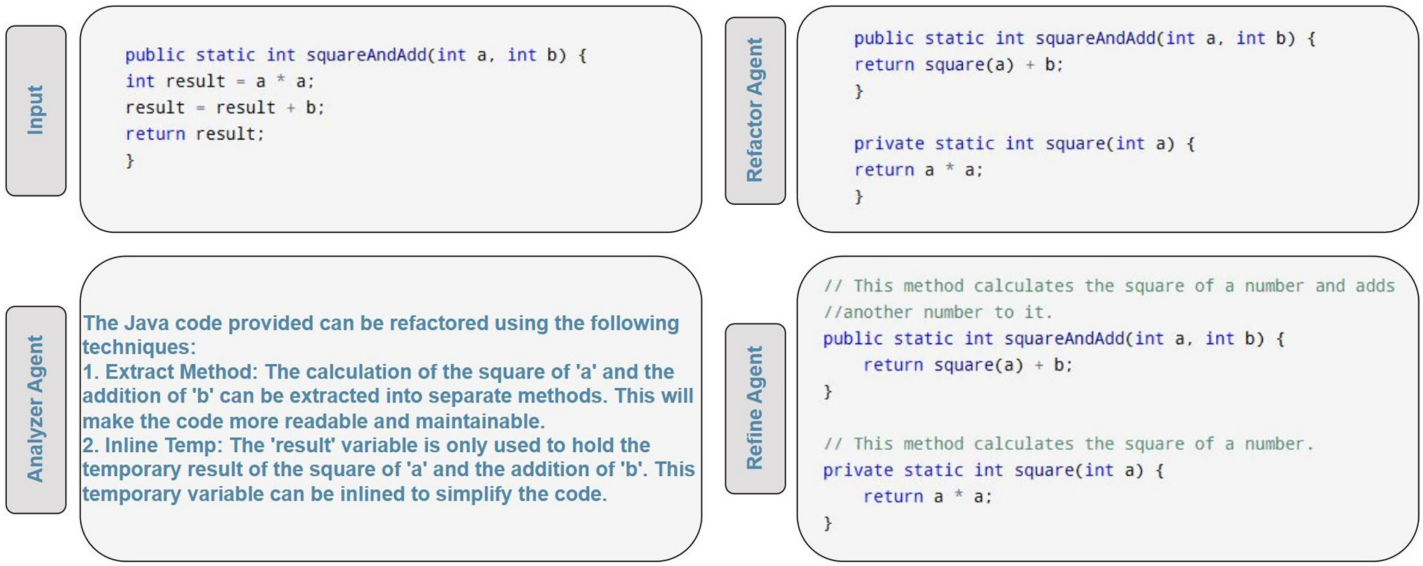

Figure 2 **Step-by-step Java code transformation through analyzer, refactor, and refine agents.** Full-size DOI: 10.7717/peerj-cs.3257/fig-2

square calculation into a dedicated method and inlining the temporary variable. Finally, the RefineAgent enhances code readability by adding concise comments to clarify method purposes. This example provides a concrete demonstration of the agent workflow described above.

The architectural design of RefactorGPT aligns with a hierarchical multi-agent pattern, where each agent operates as a specialized subprocess within a coordinated pipeline. Unlike blackboard or peer-to-peer models, RefactorGPT adopts a task-decomposition-

based orchestration, in which the output of one agent serves as the structured input to the next (*Li et al., 2024*; *de Curtò & de Zarzà, 2025*). This design ensures modularity, interpretability, and controlled delegation of responsibilities. By adhering to a top-down processing flow, the system benefits from clarity in role separation while preserving extensibility new agents or decision checkpoints can be inserted without disrupting the core pipeline. This structured interaction pattern is particularly well-suited for multi-phase reasoning tasks such as code refactoring, where each stage incrementally enhances the quality and correctness of the output.

To demonstrate its usability, the RefactorGPT framework is equipped with a lightweight web-based interface developed using Flask. This interface allows users to input raw Python or Java code and visualize each stage of the refactoring process, including the refactored, refined, and recovered versions. It also displays key performance metrics such as execution time and memory usage, providing real-time feedback on the structural and computational impact of the applied refactoring. A screenshot of the interface is shown in Fig. 3.

AnalyzerAgent

The AnalyzerAgent is responsible for determining the most appropriate refactoring technique to apply to a given code snippet. In the context of this study, the agent operates under a controlled setting: it selects from a predefined set of nine classical refactoring techniques, which aligns with how the dataset was constructed. Each code sample in the dataset was manually prepared to correspond implicitly or explicitly to one dominant technique. This constraint was introduced not due to limitations in the model's capabilities, but to enable consistent benchmarking and controlled evaluation across different technique types and difficulty levels. In broader applications, the AnalyzerAgent can be extended to operate in an unconstrained or open-ended classification mode.

The prompt issued by the agent guides the model to choose one technique from the given list and to provide a brief justification for its choice. The format of this prompt is illustrated in Fig. 4.

RefactorAgent

The RefactorAgent is the core refactoring component within the RefactorGPT framework. Its primary function is to apply the refactoring technique recommended by the AnalyzerAgent to the input code, while preserving the original functionality. Unlike typical large language model prompts that are often broad or generative in nature, the RefactorAgent operates under a tightly controlled prompt that specifies both the expected refactoring type and strict preservation of program semantics. This ensures that the refactoring output is both targeted and verifiable.

A key challenge in designing this agent lies in balancing directive clarity with generative flexibility. The prompt must be sufficiently structured to enforce the use of a particular refactoring technique such as Extract Method or Replace Temp with Query but also open enough to allow the model to restructure code naturally. Additionally, the RefactorAgent introduces an explicit performance-awareness directive: the model is instructed to refactor the code not only structurally, but with the intention of improving computational

RefactorGPT for Engineering

Paste your Python code:

```
def process_data(data_list):
    total = 0
    count = 0
    squares = []
    for item in data_list:
        if isinstance(item, int) or isinstance(item, float):
            total += item
            count += 1
            square = item * item
            squares.append(square)
        else:
            try:
                num = float(item)
```

Refactor

🔍 Analysis

1. Extract Method: The code inside the for loop can be extracted into a separate method. This will make the code easier to read and understand. The new method can be named something like `process_item`, which takes an item as input and returns the processed item and its square.

2. Split Temporary Variable: The variable `square` is used to store the square of both `item` and `num`. These can be split into two separate variables for better readability and to avoid confusion.

3. Extract Variable: The condition `isinstance(item, int) or isinstance(item, float)` can be extracted into a separate variable, say `is_number`. This will make the condition check more readable.

4. Replace Temp with Query: The `average` and `max_square` variables are used to store the results of some computations. These computations could be extracted into separate methods (or "queries"), which would make the code easier to understand and would eliminate the need for these temporary variables.

Here is a refactored version of the code:

```python
def process_data(data_list):
    squares = []
    total, count = 0, 0
    for item in data_list:
        total, count, square = process_item(item, total, count)
        if square is not None:
            squares.append(square)
    print_results(total, count, squares)

def process_item(item, total, count):
    square = None
    if isinstance(item, (int, float
```

🔧 Refactored Code

Here is the refactored code based on the analysis provided:

```python
def process_data(data_list):
    squares = []
    total, count = 0, 0
    for item in data_list:
        total, count, square = process_item(item, total, count)
        if square is not None:
            squares.append(square)
    print_results(total, count, squares)

def process_item(item, total, count):
    square = None
    is_number = isinstance(item, (int, float))
    if is_number:
        total, count, square = process_number(item, total, count)
    else:
        try:
            num = float(item)
            total, count, square = process_number(num, total, count)
        except ValueError:
            pass
```

🎯 Refine Agent Output

Enhanced Version:

```python
def process_data(data_list):
    squares = []
    total, count = 0, 0
    for item in data_list:
        total, count, square = process_item(item, total, count)
        if square is not None:
            squares.append(square)
    print_results(total, count, squares)

def process_item(item, total, count):
    square = None
    try:
        num = float(item)
        total, count, square = process_number(num, total, count)
    except ValueError:
        pass
    return total, count, square

def process_number(num, total, count):
    total += num
```

📊 Performance

	Original	Refactored	Refined	Δ
Execution Time (s)	1.0706	1.072	1.072	0.0014000000000000679
Memory Usage (KB)	36504.0	36168.0	36168.0	-336.0

Figure 3 **Web-based interface for interactive refactoring and performance monitoring.**
Full-size DOI: 10.7717/peerj-cs.3257/fig-3

AnalyzerAgent

System: You are a software refactoring expert.
User: Analyze the following Python code and recommend the most suitable refactoring technique from the following list:
- Extract Method
- Inline Method
- Inline Temp
- Replace Temp with Query
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm
- Introduce Explaining Variable

Provide the selected technique and briefly explain why it is appropriate.
Code: {code}

Figure 4 Prompt structure used by the AnalyzerAgent for refactoring technique selection.
Full-size DOI: 10.7717/peerj-cs.3257/fig-4

RefactorAgent

System: You are an expert software engineer specializing in clean code and performance-aware refactoring.
User: Refactor the following Python code according to the specified technique: Techniques.
Requirements:
- Preserve the original functionality.
- Apply the transformation cleanly and clearly.
- If possible, improve readability, modularity, and runtime or memory performance.
- Avoid introducing unnecessary changes or restructuring unrelated parts of the code.

Code: {code}

Figure 5 Prompt template for the RefactorAgent to apply the selected refactoring technique.
Full-size DOI: 10.7717/peerj-cs.3257/fig-5

efficiency where possible. This allows the resulting code to be evaluated not only for correctness, but also for potential improvements in runtime and memory usage.

The output of this agent serves as the input to downstream components (RefineAgent and FixerAgent) and is therefore required to be syntactically valid and logically coherent. A structured example of the prompt used by the RefactorAgent is presented in Fig. 5.

RefineAgent

The RefineAgent serves as the cognitive core of the RefactorGPT framework, focusing on enhancing code quality beyond the application of a specific refactoring technique. While the RefactorAgent ensures structural transformation aligned with a given pattern, the RefineAgent improves the resulting code in terms of readability, maintainability, and computational efficiency. It acts as a second-pass optimizer, simulating the role of a senior developer who reviews and polishes a proposed change.

RefineAgent

System: You are a senior software engineer specializing in code optimization and best practices.
User: Refine the following Python code to enhance its readability, maintainability, and performance. Ensure that the functionality remains unchanged.
Requirements:
- Improve code structure and modularity.
- Optimize for runtime and memory efficiency.
- Adhere to PEP 8 coding standards.
- Avoid unnecessary complexity or overengineering
Code: {code}

Figure 6 Prompt design of the RefineAgent for code quality enhancement.
Full-size DOI: 10.7717/peerj-cs.3257/fig-6

The conceptual foundation of this agent aligns with the notion of iterative refinement, a growing paradigm in language model research where outputs are recursively improved through repeated instructions. Rather than relying on informal heuristics, the RefineAgent applies a structured and goal-oriented refinement prompt designed to promote modularity, semantic clarity, and resource efficiency all while maintaining strict functional equivalence. The exact formulation of this prompt is presented in Fig. 6, illustrating how structured guidance can lead to consistent improvements in output quality.

To avoid over-refinement or redundant modifications, we empirically limited the RefineAgent to a single iteration per code example during evaluation. This decision was informed by preliminary experimentation, which showed that most measurable improvements particularly in modular structure and naming clarity were achieved in the first pass. Additional iterations occasionally led to unnecessary verbosity or deviation from the original design. However, the system architecture remains extensible and can accommodate multiple passes, potentially guided by convergence-based heuristics or user preference in future iterations.

FixerAgent

The FixerAgent was introduced as a reliability safeguard within the RefactorGPT pipeline. While the RefactorAgent and RefineAgent aim to transform code in a function-preserving manner, outputs of large language model may occasionally contain syntactic or runtime errors particularly in more complex refactoring cases. Although such failures were rare in our controlled evaluation, the inclusion of the FixerAgent ensures that the system can recover gracefully from unexpected errors, enhancing overall robustness and reinforcing confidence in the framework's practical applicability.

This agent operates by validating the transformed code through isolated execution. If an error is encountered such as a syntax error, undefined variable, or logical inconsistency the FixerAgent captures the faulty output and invokes the model with a specialized recovery prompt. This prompt instructs the model to correct the issue while strictly maintaining the intended logic of the previous refactoring. In this sense, the FixerAgent serves not as a corrective mechanism for model design flaws, but as a pragmatic solution for mitigating edge-case failures that may arise due to the stochastic nature of LLM outputs.

FixerAgent

System: You are a Python expert skilled in debugging and code correction.
User: The following Python code produces a syntax or runtime error. Please fix the error while preserving the original functionality and logic of the code.
Code: {code}

Figure 7 Recovery prompt employed by the FixerAgent for error correction.
Full-size DOI: 10.7717/peerj-cs.3257/fig-7

The prompt issued by the FixerAgent is deliberately minimal and goal focused. It does not attempt to re-refactor the code, but rather to repair it without deviating from the refactoring already applied. This makes the component lightweight, safe to invoke when needed, and compatible with both interactive and automated workflows. The FixerAgent's recovery prompt is shown in Fig. 7.

EXPERIMENTAL SETUP

This section outlines the experimental configuration used to evaluate the RefactorGPT framework. It describes the composition of the test dataset, the evaluation procedure and metrics, and the computational environment in which the experiments were conducted. Together, these elements provide a structured basis for assessing the effectiveness and reliability of each agent within the system.

Dataset description

To evaluate the effectiveness of RefactorGPT across a diverse range of transformation scenarios, we constructed two controlled datasets in Python and Java, each focusing on a wide variety of refactoring strategies. The selected techniques span a broad set of structural transformation goals in software engineering. These include Extract Method and Extract Variable, which aim to improve modularity and readability by isolating code fragments into dedicated functions or named variables (*Tsantalis & Chatzigeorgiou, 2011*; *Jiang et al., 2025*). Techniques such as Inline Method and Inline Temp perform the inverse, simplifying code by collapsing unnecessary abstractions (*Murphy-Hill, Parnin & Black, 2012*; *Oliveira et al., 2019*; *AlOmar et al., 2022*). Replace Temp with Query and Split Temporary Variable enhance clarity and side-effect isolation by promoting more transparent and purpose-specific variable usage (*Kataoka et al., 2001*; *Rongviriyapanish, Karunlanchakorn & Meananeatra, 2015*). Remove Assignments to Parameters enforces functional purity by avoiding reassignment of input arguments, while Replace Method with Method Object restructures complex procedures into cohesive, object-oriented representations (*Du Bois, Demeyer & Verelst*). Finally, Substitute Algorithm targets performance by replacing inefficient logic with more optimal implementations (*Mooij et al., 2020*). Together, these techniques provide a comprehensive benchmark for evaluating code refactoring systems across varying structural intentions.

Both the Python and Java datasets were systematically constructed by the authors to establish a controlled benchmark spanning a broad spectrum of classical refactoring

Table 1 Summary of the nine refactoring techniques used in RefactorGPT.

Refactoring technique	Description
Extract method	Moves a block of code into a new method with a meaningful name.
Inline method	Replaces a method call with the method's body when the method is trivial.
Extract variable	Introduces a new variable to hold an intermediate value for clarity.
Inline temp	Replaces a temporary variable with the expression it holds.
Replace temp with query	Substitutes a temporary variable with a method call that returns the value.
Split temporary variable	Splits a variable used for multiple purposes into separate variables.
Remove assignments to parameters	Eliminates reassignments to method parameters for safer and clearer code.
Replace method with method object	Converts a long method into an object to encapsulate and manage state.
Substitute algorithm	Replaces a low-performing algorithm with a more efficient or clearer one.

techniques. Each code sample was carefully designed to embody a single, unambiguous refactoring operation, enabling targeted evaluation of model behavior across heterogeneous transformation types. To support direct comparisons, every original example was paired with its corresponding refactored version. Each language comprises 27 examples, yielding 54 examples in total across both languages. The datasets are evenly distributed across three predefined complexity levels (easy, medium, hard), determined by structural and semantic criteria such as control-flow depth, abstraction density, and the number of variables and functions. All samples underwent a standardized preprocessing protocol to ensure syntactic validity and functional correctness, including adherence to language-specific conventions (Python 3 and Java 17), automated formatting, removal of superfluous comments, and execution-based validation for behavioral consistency. The finalized datasets were stored in structured CSV format with explicit metadata fields identifying the refactoring technique, complexity level, and original source code, thereby facilitating reproducibility and automation. The descriptions of the refactoring techniques are presented in Table 1.

Both the Python and Java datasets were systematically constructed by the authors to reflect this diversity in a controlled and reproducible fashion. Each example was intentionally crafted to represent a single, unambiguous refactoring operation, enabling precise, technique-specific evaluation of RefactorGPT's behavior. To support direct comparisons, every original example was paired with its corresponding refactored version. All samples were evenly distributed across three predefined complexity levels—easy, medium, and hard—determined by structural and semantic factors such as control-flow depth, abstraction density, and the number of variables and functions.

Prior to inclusion, each sample underwent a standardized preprocessing pipeline to ensure syntactic validity and functional correctness. This process included adherence to language-specific conventions (Python 3 and Java 17), automated formatting, removal of superfluous comments, and execution-based validation for behavioral consistency. The finalized datasets were stored in structured CSV format with explicit metadata fields identifying the refactoring technique, complexity level, and original source code, thereby supporting transparency, automation, and future extensibility.

Table 2 Distribution of refactoring techniques and difficulty levels in the evaluation dataset.

Refactoring technique	Easy		Medium		Hard	
	Java	Python	Java	Python	Java	Python
Extract method	1	1	1	1	1	1
Inline method	1	1	1	1	1	1
Extract variable	1	1	1	1	1	1
Inline temp	1	1	1	1	1	1
Replace temp with query	1	1	1	1	1	1
Split temporary variable	1	1	1	1	1	1
Remove assignments to parameters	1	1	1	1	1	1
Replace method with method object	1	1	1	1	1	1
Substitute algorithm	1	1	1	1	1	1
Total	**9**	**9**	**9**	**9**	**9**	**9**

The descriptions of the refactoring techniques are presented in Table 1. Each technique is represented by three unique code samples, one per complexity level. These difficulty levels were assigned based on structural and semantic characteristics of the code. Easy examples typically consist of short, flat code with minimal control flow and a clearly visible transformation opportunity. Medium examples introduce modest nesting, multiple variables, or auxiliary logic that requires moderate analysis to isolate refactoring targets. Hard examples feature complex or nested structures, abstract logic, and ambiguous refactoring boundaries, simulating real-world code scenarios where transformation decisions require contextual reasoning. The dataset was manually annotated to ensure that each sample contains a single dominant refactoring opportunity, making it suitable for controlled benchmarking of the RefactorGPT agents under varied structural conditions. The distribution of examples across techniques and difficulty levels is summarized in Table 2.

Evaluation protocol

Each of the 27 code samples in the dataset was independently processed through the full RefactorGPT pipeline. The system followed a sequential execution order: first, the AnalyzerAgent was used to determine the most appropriate refactoring technique; second, the RefactorAgent applied the corresponding refactoring; third, the RefineAgent enhanced the output; and finally, the FixerAgent validated and corrected any errors if present.

For each code sample, the evaluation compared three key code versions:

- The original code
- The refactored version produced by the RefactorAgent
- The refined version following additional processing by the RefineAgent

Each version was analyzed using structural and performance-based metrics, including execution time, memory usage, number of lines of code, number of defined functions, and

token count. In cases where a refactoring produced a non-executable result, the FixerAgent was invoked, and the corrected version was used for metric computation. All evaluations were conducted in isolation to ensure consistent measurement and avoid state leakage between runs.

Evaluation metrics

To assess the structural and computational impact of each refactoring, a set of five evaluation metrics was employed. These metrics were selected to capture changes in code organization, resource efficiency, and overall maintainability.

- Execution time: Measured in seconds using Python's time library, this metric reflects potential improvements or regressions in runtime performance after applying refactoring and refinement stages.
- Memory usage: Tracked in kilobytes by monitoring the peak memory consumption of each code sample during execution. The psutil library was utilized to assess whether refactoring introduced memory overhead or improved resource utilization (*Sai et al., 2024*).
- LOC: Calculated by counting the number of non-empty lines in the script. While not a direct indicator of quality, LOC offers insight into verbosity and structural simplicity fewer lines often suggest cleaner, more modular code.
- Function count: Determined by counting the number of def statements in each version of the code. This metric illustrates changes in modularity and granularity, especially in refactoring like Extract Method or Replace Method with Method Object.
- Token count: Used as a proxy for code density and cognitive complexity. Tokens comprising keywords, identifiers, operators, and whitespace were counted to estimate the cognitive load required to understand the code (*Lin et al., 2018*; *Tsantalis, Ketkar & Dig, 2022*; *Jesse, Kuhmuench & Sawant, 2023*).
- Cyclomatic complexity average (cc_avg): This metric evaluates the average number of linearly independent paths through the code, serving as a strong indicator of logical complexity and decision structure. Lower values typically correspond to simpler, more maintainable logic. The use of cc_avg is well established in software quality assessment and is particularly relevant in refactoring studies (*do Machado et al., 2014*).
- Output equivalence check: This evaluation compares the runtime output of the original and refactored code to ensure that the refactoring has not altered the program's behavior. It serves as a direct measure of functional correctness and is essential for verifying behavior preservation in automated refactoring systems.

All seven metrics were calculated for the original, refactored, and refined versions to enable consistent comparisons across stages and difficulty levels. Execution time and memory usage were measured within isolated subprocesses to ensure accurate and unbiased profiling.

Table 3 Average performance and structural metrics for Java and Python across original, refactored, and refined stages.

Metric	Original		Refactored		Refined	
	Java	Python	Java	Python	Java	Python
Execution time (ms)	1046	979	943	962	910	812
Memory usage (KB)	210	219	66	127	62	165
Lines of code	25.11	1.11	28.19	2.88	31.33	40.7
Token count	83.11	18.11	90.41	13.6	102.04	171.9
Function count	2.22	1.11	3.04	1.9	3.3	9.6
Complexity	4.22	4.22	3.63	4.03	3.26	3.85

Hardware and runtime environment

All experiments were conducted on a local machine running Windows 10, equipped with a 10-core Intel processor (Intel64 Family 6 Model 154, Stepping 4) and 32 GB of RAM. The implementation was developed and executed using Python 3.12.4. Each code refactoring and performance measurement were carried out in an isolated subprocess environment to prevent interference from background processes and to ensure consistent runtime conditions. The environment was kept constant throughout the experiments to maintain measurement reliability and reproducibility across all test cases.

RESULTS AND DISCUSSION

This section analyzes RefactorGPT's performance across the original, refactored, and refined code stages. Structural and computational metrics are reported to evaluate the contribution of each agent. Results are examined by overall trends, difficulty levels, technique-specific effects, and the corrective roles of FixerAgent and RefineAgent, offering insights into the framework's effectiveness and reliability.

Overall performance trends

To understand the overall impact of RefactorGPT on code refactoring, we used five metrics across three stages of code evolution: original, refactored, and refined. These metrics execution time, memory usage, LOC, token count, and function count capture both computational efficiency and structural complexity. The average values are summarized in Table 3.

Execution time decreases slightly from original to refined code in both languages, indicating that the refactoring stages do not introduce runtime overhead, and refinement even improves efficiency. In Java, execution time declines from 1,046 milliseconds in the original version to 910 milliseconds after refinement, while in Python it decreases from 979 milliseconds to 812 milliseconds. Memory usage remains largely stable in Python, moving from 219 kilobytes to 165 kilobytes, and shows a notable reduction in Java, from 210 kilobytes to 62 kilobytes. These results suggest that structural changes do not significantly affect resource demands and in some cases improve them.

Table 4 Average execution time and memory usage for Java and Python across difficulty levels and refactoring stages.

Difficulty	Metric	Original		Refactored		Refined	
		Java	Python	Java	Python	Java	Python
Easy	Time (ms)	1,050	1,050	943	1,040	954	1,028
	Memory (KB)	181	219	195	127	160	165
Medium	Time (ms)	1,025	1,046	954	1,033	890	1,026
	Memory (KB)	204	136	193	163	105	127
Hard	Time (ms)	1,060	1,061	931	1,033	886	1,027
	Memory (KB)	246	154	104	194	91	228

The most prominent impact is observed in structural metrics. In Java, LOC increase from 25.11 in the original version to 31.33 after refinement, and token count rises from 83.11 to 102.04. In Python, these increases are far more substantial, with LOC expanding from 1.11 to 40.7 and token count growing from 18.11 to 171.9. This pattern is also evident in function count: Java shows a modest increase from 2.22 to 3.3 on average, whereas Python exhibits a significant rise from 1.11 to 9.6. These changes indicate that the RefineAgent enhances modularity more strongly in Python by introducing explicit decomposition.

Although relative changes appear large due to the simplicity of the initial code samples, they confirm that RefactorGPT meaningfully restructures code in both languages. These effects are expected to scale with more complex codebases.

Structural and performance trends by difficulty

To assess how RefactorGPT adapts to different code complexities, we analyzed the refactoring metrics across three difficulty levels: easy, medium, and hard. These categories were manually assigned based on control flow depth, abstraction requirements, and statement density. Tables 4 and 5 present the average computational and structural metricsfor each difficulty level in Java and Python. As shown in Table 4, execution time and memory usage increase consistently with the level of difficulty in both programming languages. In Java, execution time rises from 1,050 milliseconds for easy-level original code to 1,060 milliseconds for hard-level code, while in Python it progresses from 1,050 milliseconds to 1,061 milliseconds. Memory usage follows a similar trend, with more complex examples requiring greater resources. For example, in Java, memory consumption increases from 181 kilobytes in easy-level code to 246 kilobytes in hard-level code, and in Python, from 219 kilobytes to 154 kilobytes in the original versions, with some variations after refactoring and refinement. Despite these increases, the relative differences between the original, refactored, and refined stages remain proportionally stable, indicating that the refactoring process scales effectively with complexity.

Table 5 highlights more substantial changes in structural metrics. For hard-level examples, refined code in Java shows an increase in LOC from 33.44 to 41.89 and in token count from 104.22 to 130.78, accompanied by a rise in function definitions from 2.11 to

Table 5 Average structural metrics for Java and Python across difficulty levels and refactoring stages.

Difficulty	Metric	Original		Refactored		Refined	
		Java	Python	Java	Python	Java	Python
Easy	LOC	19.33	1.11	20.89	2.11	23.89	25.56
	Tokens	63.78	12.11	66.89	8.11	81.44	95.56
	Defs	2.33	1.11	2.78	1.00	2.67	10.00
Medium	LOC	22.55	1.11	26.33	2.78	28.22	50.00
	Tokens	81.33	17.89	89.56	13.78	93.89	201.11
	Defs	2.22	1.11	3.11	1.00	3.22	10.00
Hard	LOC	33.44	1.11	37.33	3.78	41.89	47.50
	Tokens	104.22	24.33	114.78	19.00	130.78	225.00
	Defs	2.11	1.11	3.22	0.89	4	8.75

4.00. In Python, the structural changes are even more pronounced. LOC expand from 1.11 to 47.50, token count grows from 24.33 to 225.00, and function definitions increase from 1.11 to 8.75. These results indicate that RefactorGPT applies more extensive structural transformations to complex code, enhancing modularity and clarity through decomposition and the introduction of additional function definitions.

In contrast, easy-level examples show comparatively smaller structural changes. In Java, the increase in LOC from 19.33 to 23.89 and the modest rise in function count from 2.33 to 2.67 reflect limited opportunities for major restructuring. In Python, however, even easy-level code exhibits notable modularization after refinement, with function definitions increasing from 1.11 to 10.00.

Overall, these results confirm that RefactorGPT is sensitive to code complexity and adjusts the depth of refactoring accordingly. The framework preserves consistent computational performance while applying more substantial structural enhancements when greater complexity is present.

Comparison of refactor technique

To investigate how different refactoring strategies influence the structure and behavior of the refined code, we analysed the average post-refinement metrics by refactoring technique. The results are summarized in Table 6, which includes key performance and structural indicators: execution time, memory usage, LOC, token count, and function count, and cyclomatic complexity for both Java and Python.

Among all techniques, Extract Method and Extract Variable produced the most substantial structural refactoring. In Java, Extract Method yields 33.56 LOC and 116.78 tokens, with a function count of 3.11. In Python, these values are higher, reaching 36.67 LOC, 203.33 tokens, and 10 functions, confirming its strong modular decomposition effect. Extract Variable follows a similar pattern, with Java reaching 33 LOC, 107.67 tokens, and 3.89 functions, while Python attains 36.67 lines, 206.67 tokens, and 10 functions. Both techniques clearly enhance modularity and readability by isolating expressions and blocks into named functions or variables.

Table 6 Average performance and structural metrics by refactoring technique for Java and Python in the refined stage.

Refactoring techniques	Execution time (ms)		Memory usage (KB)		Lines of code		Token count		Function count		Complexity	
	Java	Python	Java	Python	Java	Python	Java	Python	Java	Python	Java	Python
Extract method	925	1,059	97	184	33.56	36.67	116.78	203.33	3.11	10.0	3.78	2.66
Inline method	844	1,026	100	250	27	23.33	78.78	110.0	2.44	10.0	3.67	4.33
Extract variable	1,015	1,029	201	94	33	36.67	107.67	206.67	3.89	10.0	4.11	5.0
Inline temp	1,020	1,013	200	108	17.33	20.0	57	70.0	2.11	10.0	3.56	4.0
Replace temp with query	1,016	1,022	268	197	29.89	30.0	96.11	160.0	3	10.0	3.22	3.66
Split temporary variable	993	1,043	101	122	26.11	33.33	89.44	130.0	3.11	10.0	3.56	3.66
Remove assignments to parameters	1,007	1,017	44	261	19.78	20.0	71.11	86.67	2.22	6.67	2.22	2.0
Replace method with method object	989	1,017	100	138	29.67	73.33	101.44	246.67	2.89	10.0	3.11	3.0
Substitute algorithm	886	1,009	106	170	37.56	86.67	108.33	300.0	2.89	10.0	6.11	5.66

Techniques such as Inline Method and Inline Temp show more modest structural impact. In Java, Inline Method produces 27 LOC and 78.78 tokens, while in Python the corresponding values are 23.33 lines and 110 tokens. Function counts remain low in Java but reach 10 in Python, reflecting language-specific tendencies in modularization. Inline Temp results are similar, with comparatively lower LOC and token counts, aligning with their minimal refactoring goals.

Replace Temp with Query and Split Temporary Variable lead to moderate increases in structural measures. For example, in Python, Replace Temp with Query reaches 160 tokens and 10 functions, while Split Temporary Variable produces 130 tokens and 10 functions. These results reflect clearer expression semantics and function-level encapsulation.

Remove Assignments to Parameters stands out due to its low function counts in Java, with 2.22 functions, and in Python, with 6.67 functions, yet higher memory usage in Python, reaching 261 kilobytes. This may be attributed to duplicated variable assignments or function overheads introduced during refinement.

Replace Method with Method Object demonstrates stronger structural changes in Python, where LOC rise to 73.33 and tokens to 246.67, accompanied by 10 functions. Java values are more moderate, with 29.67 lines, 101.44 tokens, and 2.89 functions.

Substitute Algorithm, although intended to improve performance, does not show notable reductions in execution time for these relatively small code samples. However, in Python, it results in substantial structural expansion, with 86.67 LOC, 300 tokens, and 10 functions, while in Java it maintains 37.56 lines, 108.33 tokens, and 2.89 functions.

Overall, the comparison indicates that techniques aimed at clarity and modularity, such as Extract Method and Extract Variable, yield the most significant structural transformations, especially in Python. Performance-oriented techniques, while potentially impactful on larger-scale code, show limited observable runtime benefits in this controlled setting. These findings suggest that RefactorGPT effectively aligns its refactoring with the intended goals of each technique and adapts its structural changes to the characteristics of the target language.

FixerAgent performance

Although the primary focus of this study was on code refactoring and refinement quality, ensuring syntactic and runtime correctness remained a critical concern. In early testing, we observed that approximately 10% of refactored code samples even when generated by a high-performing model produced errors upon execution due to issues such as missing return statements, improper indentation, or unclosed code blocks.

To address this, a FixerAgent was integrated into the RefactorGPT pipeline. This agent automatically scanned the refactored output and corrected common syntactic errors or formatting inconsistencies, often using minimal edits to preserve the intent of the original refactoring. The FixerAgent was invoked selectively, only when an error was detected during runtime testing.

Following its inclusion, the system achieved 100% reliability in resolving runtime execution errors, although functional output discrepancies were addressed separately and are reported in 'Evaluating Output Consistency Across Refactor Stages'. While FixerAgent's corrections were not analyzed in detail in this study, its utility in maintaining pipeline stability was indispensable. Future work may investigate the nature and frequency of the errors it resolves, offering further insights into the post-processing needs of LLM-based code generation.

Impact of RefineAgent

The RefineAgent is designed to enhance the output of the initial refactoring by further improving code readability, modularity, and overall structure. To assess its impact, we compared the refined code against the refactored version using five core metrics: execution time, memory usage, LOC, token count, and function count.

The results, summarized in Table 2, reveal that RefineAgent consistently increases the structural richness of the code. On average, refined versions contain significantly more tokens and functions, and the LOC value rises markedly. These increases are not indicative of redundancy, but rather reflect the insertion of modular structures, clearer logic segmentation, and better-named abstractions. For example, the average number of functions rises from 0.96 in the refactored stage to 9.62 in the refined code highlighting the agent's role in promoting decomposition and abstraction.

In contrast, computational performance remains relatively stable. Execution time shows a slight improvement, while memory usage increases marginally. These fluctuations are minimal compared to the structural gains and suggest that the agent's refactoring preserve operational efficiency while enhancing maintainability.

Overall, the RefineAgent demonstrates meaningful improvements in code structure without imposing significant computational overhead. Its contributions are especially valuable in complex scenarios, where initial refactoring may not fully capture the semantic intent or modular potential of the input code.

Comparison with existing approaches

Table 7 presents a comparative analysis between RefactorGPT and several recent refactoring approaches from the literature. EM-Assist integrates LLM recommendations

Table 7 Comparative analysis of RefactorGPT and existing refactoring approaches regarding methodology, scope, safety mechanisms, and evaluation metrics.

Study	Approach type	Scope (Techniques and languages)	Safe application	Evaluation metrics
EM-Assist (*Pomian et al., 2024* ICSME)	LLM + IDE integration	Single technique (Extract Method), Java	Guaranteed by IDE refactoring engine	User study, quality ratings, public dataset
Liu et al. (2024a) TOSEM	Empirical evaluation of LLMs for refactoring	Multiple opportunities, Java (20 projects)	Reapplied *via* IDE ("RefactoringMirror"), semantic validation	Success rate, safety ratio, public dataset
DePalma et al. (2024) ESWA	Empirical study of ChatGPT's refactoring capabilities	Multiple ad-hoc examples, Java/ Python/C#	Manual inspection, partial validation	Qualitative assessment
Zhang et al. (2024)	Hybrid knowledge-based + LLM	Pythonic idiomatic transformations, Python	Style validation	Style/idiom conformity metrics
RefactorGPT	Multi-agent LLM framework + FixerAgent + Web-based user interface	Nine classical refactoring techniques, Python & Java	Automated execution tests with FixerAgent-based recovery	Execution time, memory usage, LOC, functions, tokens, cc_avg, output check, public datasets

with IDE capabilities but is limited to a single refactoring technique (Extract Method) and one programming language (Java) (*Pomian et al., 2024*). Empirical evaluations of LLM-based refactoring have also been conducted, yet these studies focus primarily on assessment rather than offering an automated, multi-agent pipeline with built-in error recovery (*Liu et al., 2024b*; *DePalma et al., 2024*). A hybrid knowledge-based and LLM approach has been used to target stylistic improvements in Python, but this work does not address structural refactoring across multiple languages (*Zhang et al., 2024*). In contrast, RefactorGPT combines a multi-agent architecture, a FixerAgent for automated error handling, and a web-based user interface to support nine classical refactoring techniques in both Python and Java. Moreover, it ensures safe application through automated execution tests and output verification, and it evaluates performance using a diverse set of metrics, including cc_avg and output equivalence, while making datasets and code publicly available for reproducibility.

Evaluating output consistency across refactor stages

To assess the functional correctness of the refactoring pipeline, an output equivalence check was applied across all 54 experimental cases. This evaluation revealed that eight of the refactored codes and six of the refined codes produced outputs that deviated from the original, indicating functional inconsistencies introduced during transformation. In total, 14 instances failed to preserve output behavior at either the refactoring or refinement stage. These failures were subsequently addressed by the FixerAgent, which was responsible for analyzing and correcting both runtime errors and semantic mismatches. The FixerAgent successfully resolved 18 faulty cases, including those with execution errors and output discrepancies. However, despite the sequential processing through all four agents, 6 of the final code versions still failed the output equivalence check. This outcome highlights both

the corrective potential and the remaining limitations of the system in achieving fully reliable, output-preserving refactoring.

CONCLUSIONS

This study introduced RefactorGPT, a modular multi-agent framework designed to automate code refactoring through interactions with ChatGPT. The system leverages four specialized agents AnalyzerAgent, RefactorAgent, RefineAgent, and FixerAgent each responsible for a distinct subtask in the refactoring pipeline. Unlike monolithic prompting approaches, RefactorGPT enables structured, traceable, and extensible refactoring through agent-driven decomposition.

Experimental results across 27 code samples demonstrated the system's effectiveness in improving structural properties such as modularity, readability, and decomposition. Metrics including token count, function count, and LOC consistently increased after refinement, indicating more explicit and maintainable code. RefactorGPT also showed sensitivity to code complexity, with deeper refactoring applied to more challenging examples. Among the techniques tested, Extract Method and Extract Variable contributed the most substantial structural enhancements, aligning with their modular design goals.

Overall, RefactorGPT provides a structured, multi-agent framework that coordinates prompt-driven interactions to perform code refactoring tasks using large language models. By modeling code transformation as a collaborative pipeline and embedding verification mechanisms at various stages, the system balances creativity and correctness while enhancing transparency through intermediate artifacts. The results demonstrate its ability to operate on both Python and Java code with consistent output-preserving transformations across complexity levels and refactoring techniques.

Despite the promising results and the flexible architecture of RefactorGPT, several limitations remain. First, the experiments were conducted on a controlled dataset rather than using large-scale open-source repositories, which may limit the generalizability of the findings. Second, although the proposed framework demonstrates high compatibility with Java, additional testing on other languages is necessary to confirm its adaptability across diverse programming ecosystems. Third, while the study incorporates metrics like cyclomatic complexity and output equivalence, other internal quality metrics such as coupling and cohesion were not included in the current evaluation and are left for future exploration. Addressing these limitations in subsequent studies will further strengthen the applicability and robustness of the proposed framework.

ACKNOWLEDGEMENTS

The authors acknowledge the use of ChatGPT (OpenAI) for language editing and clarity enhancement during manuscript preparation. All scientific content, analyses, and conclusions were developed and verified by the authors.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Muhammed Abdulhamid Karabiyik conceived and designed the experiments, performed the experiments, analyzed the data, performed the computation work, prepared figures and/or tables, authored or reviewed drafts of the article, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

The code and data is available at Zenodo: KARABIYIK, M. A. (2025). RefactorGPT Code and dataset files. Zenodo. https://doi.org/10.5281/zenodo.16782314.

Supplemental Information

Supplemental information for this article can be found online at http://dx.doi.org/10.7717/peerj-cs.3257#supplemental-information.

REFERENCES

Akilesh S, Sekar R, Om Kumar CU, Prakalya D, Suguna S. 2025. Multi-agent hierarchical workflow for autonomous code generation with large language models. In: *2025 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*. Piscataway: IEEE DOI 10.1109/SCEECS64059.2025.10940635.

AlOmar EA, Liu J, Addo K, Mkaouer MW, Newman C, Ouni A, Yu Z. 2022. On the documentation of refactoring types. *Automated Software Engineering* **29(1)**:9 DOI 10.48550/arXiv.2112.01581.

Bekefi BF, Szabados K, Kovacs A. 2019. A case study on the effects and limitations of refactoring. In: *2019 IEEE 15th International Scientific Conference on Informatics*. Piscataway: IEEE DOI 10.1109/Informatics47936.2019.9119321.

Buse RPL, Weimer WR. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering* **36(4)**:546–558 DOI 10.1109/TSE.2009.70.

de Curtò J, de Zarzà I. 2025. LLM-driven social influence for cooperative behavior in multi-agent systems. *IEEE Access* **13**:44330–44342 DOI 10.1109/ACCESS.2025.3548451.

De Smedt T, Daelemans W. 2012. Pattern for python. *Journal of Machine Learning Research* **13**:2063–2067.

DePalma K, Miminoshvili I, Henselder C, Moss K, AlOmar EA. 2024. Exploring ChatGPT's code refactoring capabilities: an empirical study. *Expert Systems with Applications* **249**:123602 DOI 10.1016/j.eswa.2024.123602.

do Machado IC, McGregor JD, Cavalcanti YC, de Almeida ES. 2014. On strategies for testing software product lines: a systematic literature review. *Information and Software Technology* **56(10)**:1183–1199 DOI 10.1016/j.infsof.2014.04.002.

Du Bois B, Demeyer S, Verelst J. Refactoring—improving coupling and cohesion of existing code. In: *11th Working Conference on Reverse Engineering*. IEEE Computer Society, 144–151 DOI 10.1109/WCRE.2004.33.

Hemel Z, Kats LCL, Groenewegen DM, Visser E. 2010. Code generation by model transformation: a case study in transformation modularity. *Software & Systems Modeling* **9(3)**:375–402 DOI 10.1007/s10270-009-0136-1.

Hou Z, Tang J, Wang Y. 2025. HALO: hierarchical autonomous logic-oriented orchestration for multi-agent LLM systems. ArXiv DOI 10.48550/arXiv.2505.13516.

Jesse K, Kuhmuench C, Sawant A. 2023. RefactorScore: evaluating refactor prone code. *IEEE Transactions on Software Engineering* **49(11)**:5008–5026 DOI 10.1109/TSE.2023.3324613.

Jiang Y, Chi X, Zhang Y, Ji W, Li G, Wang W, Xia Y, Zhang L, Liu H. 2025. Automated recommendation of extracting local variable refactorings. *ACM Transactions on Software Engineering and Methodology* **29**:28 DOI 10.1145/3715110.

Kataoka Y, Ernst MD, Griswold WG, Notkin D. 2001. Automated support for program refactoring using invariants. In: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. Piscataway: IEEE, 736–743 DOI 10.1109/ICSM.2001.972794.

Li X, Wang S, Zeng S, Wu Y, Yang Y. 2024. A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth* **1(1)**:9.

Lin B, Zampetti F, Bavota G, Di Penta M, Lanza M, Oliveto R. 2018. Sentiment analysis for software engineering. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 94–104 DOI 10.1145/3180155.3180195.

Liu Y. 2025. Attention is all large language model need. *ITM Web of Conferences* **73(11)**:02025 DOI 10.1051/itmconf/20257302025.

Liu B, Jiang Y, Zhang Y, Niu N, Li G, Liu H. 2024a. An empirical study on the potential of LLMs in automated software refactoring. ArXiv DOI 10.48550/arXiv.2411.04444.

Liu Y, Le-Cong T, Widyasari R, Tantithamthavorn C, Li L, Le X-BD, Lo D. 2024b. Refining ChatGPT-generated code: characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology* **33(5)**:1–26 DOI 10.1145/3643674.

Mooij AJ, Ketema J, Klusener S, Schuts M. 2020. Reducing code complexity through code refactoring and model-based rejuvenation. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Piscataway: IEEE, 617–621 DOI 10.1109/SANER48275.2020.9054823.

Murphy-Hill E, Parnin C, Black AP. 2012. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* **38(1)**:5–18 DOI 10.1109/TSE.2011.41.

Oliveira J, Gheyi R, Mongiovi M, Soares G, Ribeiro M, Garcia A. 2019. Revisiting the refactoring mechanics. *Information and Software Technology* **110(2)**:136–138 DOI 10.1016/j.infsof.2019.03.002.

Pomian D, Bellur A, Dilhara M, Kurbatova Z, Bogomolov E, Bryksin T, Dig D. 2024. Next-generation refactoring: combining LLM insights and IDE capabilities for extract method. In: *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Piscataway: IEEE, 275–287 DOI 10.1109/ICSME58944.2024.00034.

Prinz R. 2022. The modularity codes. *Biosystems* **219(7)**:104735 DOI 10.1016/j.biosystems.2022.104735.

Rochimah S, Arifiani S, Insanittaqwa VF. 2015. Non-source code refactoring: a systematic literature review. *International Journal of Software Engineering and Its Applications* **9(6)**:197–214.

Rongviriyapanish S, Karunlanchakorn N, Meananeatra P. 2015. Automatic code locations identification for replacing temporary variable with query method. In: *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. Piscataway: IEEE DOI 10.1109/ECTICon.2015.7207086.

Sai PC, Karthik K, Prasad KB, Sai Pranav CV, Divya KV. 2024. Real-time task manager: a python-based approach using psutil and tkinter. In: *2024 8th International Conference on Computational System and Information Technology for Sustainable Solutions (CSITSS)*. Piscataway: IEEE DOI 10.1109/CSITSS64042.2024.10816758.

Sanderson K. 2023. GPT-4 is here: what scientists think. *Nature* **615(7954)**:773 DOI 10.1038/d41586-023-00816-5.

Shirafuji A, Oda Y, Suzuki J, Morishita M, Watanobe Y. 2023. Refactoring programs using large language models with few-shot examples. In: *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. Piscataway: IEEE, 151–160 DOI 10.1109/APSEC60848.2023.00025.

Tsantalis N, Chatzigeorgiou A. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* **84(10)**:1757–1782 DOI 10.1016/j.jss.2011.05.016.

Tsantalis N, Ketkar A, Dig D. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* **48(3)**:930–950 DOI 10.1109/TSE.2020.3007722.

Wei Y, Xia CS, Zhang L. 2023. Copiloting the copilots: fusing large language models with completion engines for automated program repair. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 172–184 DOI 10.1145/3611643.3616271.

Zhang Z, Xing Z, Ren X, Lu Q, Xu X. 2024. Refactoring to pythonic idioms: a hybrid knowledge-driven approach leveraging large language models. *Proceedings of the ACM on Software Engineering* **1(FSE)**:1107–1128 DOI 10.1145/3643776.