future internet

MDPI

Article

Refactoring Loops in the Era of LLMs: A Comprehensive Study

Alessandro Midolo and Emiliano Tramontana *

Dipartimento di Matematica e Informatica, University of Catania, 95125 Catania, Italy; alessandro.midolo@unict.it
* Correspondence: tramontana@dmi.unict.it; Tel.: +39-095-7383008

Abstract

Java 8 brought functional programming to the Java language and library, enabling more expressive and concise code to replace loops by using streams. Despite such advantages, for-loops remain prevalent in current codebases as the transition to the functional paradigm requires a significant shift in the developer mindset. Traditional approaches for assisting refactoring loops into streams check a set of strict preconditions to ensure correct transformation, hence limiting their applicability. Conversely, generative artificial intelligence (AI), particularly ChatGPT, is a promising tool for automating software engineering tasks, including refactoring. While prior studies examined ChatGPT's assistance in various development contexts, none have specifically investigated its ability to refactor for-loops into streams. This paper addresses such a gap by evaluating ChatGPT's effectiveness in transforming loops into streams. We analyzed 2132 loops extracted from four open-source GitHub repositories and classified them according to traditional refactoring templates and preconditions. We then tasked ChatGPT with the refactoring of such loops and evaluated the correctness and quality of the generated code. Our findings revealed that ChatGPT could successfully refactor many more loops than traditional approaches, although it struggled with complex control flows and implicit dependencies. This study provides new insights into the strengths and limitations of ChatGPT in loop-to-stream refactoring and outlines potential improvements for future AI-driven refactoring tools.

Keywords: AI-based code; code generation; software engineering

Academic Editors: Diego Vergara and Pablo Fernández-Arias

Received: 2 August 2025
Revised: 9 September 2025
Accepted: 10 September 2025
Published: 12 September 2025

Citation: Midolo, A.; Tramontana, E. Refactoring Loops in the Era of LLMs: A Comprehensive Study. *Future Internet* **2025**, *17*, 418. https://doi.org/10.3390/fi17090418

1. Introduction

The release of Java 8 introduced the functional programming paradigm in the Java language and library and enabled developers to implement more expressive and concise code. The functional paradigm reduces common programming errors, such as off-by-one mistakes, while promoting higher-level abstraction thanks to constructs like streams and lambda expressions [1,2], while the underlying libraries handle lower level details [3]. Despite the significant advantages offered by Java 8, traditional loops, particularly the for-loop, are still prevalent in available codebases [4]. The transition to a functional paradigm requires a change in mindset, which many developers have not yet fully adopted [5,6].

Generative AI technologies, such as ChatGPT, provide a very powerful coding assistant that can solve many software engineering problems, especially in the realm of code refactoring [7,8]. ChatGPT, a generative pre-trained transformer (GPT), allows users to input prompts and receive human-like responses across various media, including text and code [9]. Given the critical importance of refactoring [10,11], and the many automated tools proposed in the literature [12–16], the potential of ChatGPT to improve code quality and development efficiency deserves closer examination [17].

Researching ChatGPT's refactoring ability gives an opportunity to assess both the code quality and the efficiency of the development process in terms of time and cost. Moreover, ChatGPT's ability to maintain context across multiple conversational exchanges and generate human-like dialogue offers unique advantages not found in other AI-based tools. Therefore, a thorough investigation of ChatGPT's capabilities in this domain could lead to significant advancements in software engineering.

Several studies have explored the role of ChatGPT in software engineering. Depalma et al. [8] examined ChatGPT's effectiveness in refactoring Java code, highlighting its strengths in improving code readability but suggesting the need for human oversight to maintain broader context and ensure correctness. White et al. [18] proposed a set of prompt patterns designed to leverage ChatGPT and other large language models (LLMs) for common software engineering tasks. Chavan et al. [19] analyzed the outcomes of ChatGPT on software development by studying the interactions on GitHub and Hacker News and focusing on its role in refactoring. Guo et al. [20] compared ChatGPT with the CodeReviewer tool, finding that ChatGPT performs better in specific code review tasks. Other studies [21–23] examined the effectiveness of ChatGPT in various development tasks and identified strengths and limitations in several contexts.

Although prior studies have examined various uses of ChatGPT in software engineering, none have specifically addressed the task of refactoring for-loops into stream pipelines. This gap presents a significant opportunity for further research. Investigating ChatGPT's suggestions for transitioning from an imperative to a functional programming style could provide valuable insights into ChatGPT's abilities. The literature describes two main deterministic approaches for refactoring loops into stream pipelines [12,24]. The more recent approach [12] introduced five templates for identifying and refactoring for-loops into streams, while the foundational work by Gyori et al. [24] explored multiple transformations. However, the applicability of both methods depends on the satisfaction of a strict set of preconditions that guarantees the correct transformation of each category of loops. More loops could be refactored if more categories or more sets of preconditions could be discovered.

To address such limitations, this paper proposes a comprehensive study that leverages ChatGPT's ability to refactor for-loops into stream pipelines. The study aims to (i) evaluate ChatGPT's effectiveness in preserving the semantics of code during refactoring, (ii) compare ChatGPT's performance to state-of-the-art (SOTA) approaches by analyzing whether it can overcome the limitations inherent in other approaches, and (iii) identify and analyze the scenarios in which ChatGPT fails to refactor code and explore the causes of failures.

For this study, we extracted five open source repositories on GitHub and isolated 2132 loops. Each loop was labeled according to the preconditions and templates defined by the SOTA approaches. The code fragments were then submitted to ChatGPT, which was tasked with generating a refactored version using stream pipelines. We provide a detailed discussion of the typical use cases for each label, highlight successful refactoring cases, and analyze the instances where ChatGPT's output was either flawed or counterproductive. The replication package of our experiments is publicly available [25].

The rest of the paper is structured as follows. In Section 2, we review the current state of the art on loop refactoring and the application of ChatGPT in software engineering. Section 3 details the methodology of our study, including its objectives and execution. Section 4 presents the results, discusses the implications, and outlines the limitations of ChatGPT's performance. Section 5 addresses potential threats to the validity of our study. Finally, Section 6 concludes with a summary of our contributions and suggestions for future research directions.

2. Related Works

This section reviews the main related works using LLMs and recommender systems for software development activities. Specifically, we focus on studies that (i) propose various approaches to refactoring for-loops and (ii) explore the use of ChatGPT in software-related tasks.

2.1. For-Loop Refactoring

The literature suggests two main approaches for refactoring for-loops into appropriate stream code. The first study introduces LambdaFicator [15], a tool designed to automate refactoring tasks in Java by leveraging functional features introduced in Java 8 [24]. LambdaFicator performs two key refactorings: converting anonymous inner classes to lambda expressions and transforming for-loops that iterate over collections into functional operations (like map or filter) using lambda expressions.

The second approach automatically refactors for-loops into stream-based code in Java, leveraging the functional programming features introduced in Java 8 [12]. The approach works by checking for-loop code against preconditions and matching it to predefined templates that represent equivalent stream-based operations. Once a match is found, the loop is replaced with the corresponding stream code, adapting and incorporating relevant statements from the original loop. Both approaches define a set of preconditions that guide the refactoring by discarding all for-loops that do not satisfy at least one of them. This paper focuses on the preconditions and investigates the ability of ChatGPT to overcome the limits imposed by preconditions and proposes innovative refactoring solutions.

Other approaches focused on refactoring for-loops, by optimizing their structure and execution or proposing parallel solutions to improve their efficiency. In [26], the authors introduced a technique for summarizing memoryless loops in C programs—loops that handle strings without maintaining memory across iterations—by replacing them with equivalent C standard library functions. This counterexample-guided synthesis approach aims to simplify string manipulation, which is often achieved through complex, error-prone loops in C.

One of the key advantages of the Java Stream API is the ability to easily parallelize streams using the parallel() method or by creating a stream using the parallelStream() method. However, before parallelizing stream operations, several factors must be considered as they may negatively impact performance. Khatchadourian et al. [27] analyzed stream operations to determine whether they are suitable for parallelization. They defined a set of conditions to ensure that converting sequential operations to parallel ones will enhance performance. Their approach is applicable to projects that include at least one stream declaration. Given the high-level abstractions provided by the Stream API, it can be challenging for developers to account for control and data flow relationships, particularly in the context of parallel operations [28]. To address this, Stein et al. [29] proposed an analysis of the user interface (UI) access in stream-based programs to ensure thread safety in parallelized environments.

Python (version 3.11.6) follows a set of conventions and best practices, known as Python Enhancement Proposals (PEPs) or Pythonic idioms [30], to enhance expressiveness and conciseness. Among these, some Pythonic idioms enable the refactoring of for-loops into a functional style. Zhang et al. [13,14] introduced a static analysis tool that automatically refactors anti-idiomatic code smells into Pythonic idioms. Similarly, Midolo et al. [31] evaluated the effectiveness of LLMs, specifically GPT-4o, in refactoring such code smells. Their comparison with the state-of-the-art tool [14] demonstrated that LLMs achieve comparable or superior performance.

2.2. AI for Software Engineering

In [8], the authors investigated ChatGPT's abilities and limitations in refactoring Java code, focusing on its capacity to improve code quality while preserving the behavior and providing accurate documentation. The study evaluates ChatGPT's performance on 40 Java code segments based on eight quality attributes. The analyzed for-loop refactoring focused solely on transforming traditional for-loops into enhanced for-loops and removing inefficient operations. Conversely, our approach investigates the ability of ChatGPT to refactor for-loop to stream pipelines, moving from an imperative style to a functional one.

A recent study investigated the effectiveness of general-purpose LLMs in automating software refactoring [32]. Through an empirical evaluation, the researchers assessed LLMs on a range of refactoring tasks, measuring both the correctness of the transformations and their impact on code quality metrics. The results showed that while LLMs can successfully suggest many refactored code snippets, challenges remain in preserving program semantics and handling complex or large codebases.

Another empirical analysis focused on StarCoder2's ability to perform code refactoring on 30 open-source Java projects [33]. The findings indicated that the model can reduce code smells by 20.1%, more than human developers on average, particularly excelling at systematic issues such as long statements, magic numbers, and empty catch clauses. However, developers still outperform the model in more complex design-level problems such as broken modularization and deficient encapsulation.

The EM-Assist tool has been proposed as an IntelliJ IDEA plugin that enhances the traditional Extract Method refactoring process by combining LLM-based reasoning with IDE-supported static analysis [34]. This approach addresses the challenge of identifying which statements in overly long methods should be extracted—a task where existing tools often fall short of matching developers' preferences.

An investigation into developer–ChatGPT interactions provided insights into how programmers use LLMs to support refactoring tasks [35]. By analyzing real refactoring conversations, the study highlights the prompts developers use, the types of refactoring suggested, and how developers respond—accepting, modifying, or rejecting the AI's recommendations. Common patterns emerge, alongside challenges such as potential behavioral changes in the code, incomplete understanding of developer intent, and trust issues.

White et al. [18] introduced prompt design techniques, structured as patterns, to address common challenges when using LLMs, e.g., ChatGPT, in software engineering tasks. These tasks include ensuring code decoupling from third-party libraries and generating API specifications from requirement lists.

In [19], an investigation of the impact of ChatGPT on software development was conducted to analyze developer conversations from GitHub and Hacker News. Through exploratory data analysis and data annotation, the study characterized interactions between developers and ChatGPT, with a specific focus on its role in code refactoring. By filtering relevant keywords, the researchers identified and analyzed refactoring facilitated by ChatGPT.

A study evaluating ChatGPT's performance in automated code review tasks was presented in [20]. The approach focused on the ability of ChatGPT to refine code based on the provided reviews. Using the existing CodeReview benchmark and a newly constructed dataset, the authors compared ChatGPT's performance against CodeReviewer, a state-of-the-art tool. The results show that ChatGPT significantly outperforms CodeReviewer.

The study in [21] examined the practical use of ChatGPT by developers through an analysis of the DevGPT dataset, which contains developer–ChatGPT conversations, including code snippets. The study addresses three key research questions: (1) whether ChatGPT generates Python and Java code with quality issues, (2) whether ChatGPT-generated code

is merged into repositories and how much it is modified, and (3) the main use cases for ChatGPT beyond code generation. The findings reveal that ChatGPT-generated code often contains issues such as undefined variables, poor documentation, and security flaws.

The study in [22] explored software development practices by mining 1501 commits, pull requests (PRs), and issues from open-source projects to identify instances where ChatGPT was used. After filtering out false positives, the authors manually analyzed 467 true positive cases, resulting in a taxonomy of 45 tasks that developers automated by means of ChatGPT. These tasks, categorized and illustrated with representative examples, provided valuable insights for developers on how to integrate LLMs into their workflows and offer researchers a clear view of tasks that could benefit from further automation.

In [23], the authors systematically compared the unit test suites generated by ChatGPT and search-based software testing (SBST) tool EvoSuite [36]. The comparison focused on key aspects such as correctness, readability, code coverage, and bug detection capability. The study identified the strengths and weaknesses of ChatGPT in generating unit tests and its effectiveness against EvoSuite.

These studies explored various aspects of the usage of ChatGPT in software engineering tasks, highlighting both its strengths and weaknesses when assisting developers. However, the capability of ChatGPT to refactor for-loops into semantically equivalent stream pipelines has not been previously investigated. This gap indicates a need for further research into the model's ability to facilitate the transition from imperative programming styles to functional programming paradigms.

3. Investigation Approach

3.1. Java Streams: A Functional Alternative to Looping

Java streams, introduced in Java 8, offer a powerful and expressive alternative to traditional for-loops, enabling developers to process collections with a functional approach. While for-loops provide means to implement the iteration and the statements to execute over a set of elements, streams provide a way to implement a more concise and readable code, reducing boilerplate and improving maintainability. Streams support lazy evaluation, meaning that operations are executed only when a terminal operation is invoked, ensuring efficiency by avoiding unnecessary computations.

Listing 1 shows an example of a typical for-loop that iterates over a collection, requiring explicit control of the loop counter and conditions. This imperative style requires developers to manually specify the iteration, condition checking, and the actions to perform, which leads to verbose code, especially when having to perform complex transformations.

Listing 1. A traditional for-loop selecting from a list and printing some values.

```
1 List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
2 for (String name : names) {
3     if (name.startsWith("A")) {
4         System.out.println(name.toUpperCase());
5     }
6 }
```

Listing 2 shows the corresponding Java stream-based code. Here, the stream pipeline consists of a series of operations, including filtering and mapping, where each step clearly expresses its purpose. The filter() method allows an element of the stream to be checked according to the given condition and only provides as output the elements that match the condition; whereas the map() method transforms the elements according to the given operation. As a result, the code is more readable and maintainable.

Listing 2. A stream pipeline used to select values and print them.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream()
     .filter(name -> name.startsWith("A"))
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

The declarative nature of Java streams offers several significant benefits. First, streams improve conciseness. Where traditional loops require multiple lines of code to iterate, check conditions, and transform elements, streams reduce this to a single line per operation. The removal of boilerplate code allows developers to express complex logic in a much clearer and more compact way. Second, readability is greatly improved. By chaining operations, such as filter() and map(), the intent of the code is immediately apparent. This contrasts with traditional loops, where the mechanics of iteration can obscure the underlying logic. As a result, streams make it easier to understand what the code is doing at a glance.

Another advantage of streams is their inherent support for parallelism. Traditional loops require manual implementation of multithreading, often involving complex thread management. Streams, instead, offer built-in parallel processing capabilities by means of the parallelStream() method. This allows developers to efficiently distribute tasks across multiple processor cores, significantly improving performance in scenarios involving large datasets. Lastly, streams align with Java's broader move towards functional programming, introducing a more functional style through the use of lambdas and method references. This shift encourages developers to think in terms of operations on datasets rather than focusing on individual elements and control structures.

The current work aims to assess the power and reliability of ChatGPT to transform for-loops. This in turn provides developers with suggestions on whether complex for-loops could be transformed using ChatGPT, and whether refactoring is possible even though not documented in the literature. This knowledge is valuable for developers who wish to improve their code or implement new code.

3.2. Solutions and Limits of the State of the Art

We focus on the limits of SOTA approaches. In particular, we have analyzed the preconditions expressed in LambdaFicator [15,24] and in our previous work [12]. In both works, the preconditions serve to check the functional requisites of the for statements before the refactoring. If a for-loop does not satisfy any one precondition, it is not considered for the refactoring. Table 1 gives the main characteristics of the code that allow the loop to be labeled according to a category and the preconditions that have to be satisfied for such a category to be able to transform the loop. Table 2 shows each transformation template with the relevant label and description. These are the transformations proposed by both approaches above, which are apt to for-loops that satisfy the previous preconditions, and additionally a set of five templates presented in [12].

Table 1. Categories of loops and preconditions to allow refactoring defined by the approaches in [15,24].

Name	Label	Description	Precondition
Local Variables	LV	References to non-effectively final variables defined outside the loop	Only one reference allowed
Throw Clause	TC	Throwing checked exceptions	No throws allowed
Break	BR	Containing break statements	No break allowed

Table 1. *Cont.*

Name	Label	Description	Precondition
More Return	MR	Containing return statements	Only one return allowed
Continue	CO	Containing a continue statement	No continue allowed
Max Statements	MS	Number of statements	At most five statements (in if-then-else both branches are considered as one statement)
Nested Loop	NL	Containing nested loops	No nested loops allowed
Switch	SW	Containing switch statements	No switch statements allowed

Table 2. Categories of loops and templates suggested for the refactoring according to [12].

Name	Label	Description
One return	OR	The loop contains one return statement
Small size	SS	The loop has at most five statements
First template	1st	Loops that contain a conditional statement and one or more return statements giving back a primitive value or an object
Second template	2nd	Loops having a temp variable [10] declaration and assignment then a conditional statement whose expression uses the assigned variable and a return statement using the assigned variable
Third template	3rd	Loops having a temp variable and some other statements, such as a conditional statement using the current value of the iteration, and a method call
Forth template	4th	Loops where some operation is called on each element of the iteration to create a new value, then such a new value is inserted into a collection
Fifth template	5th	Loops having an if-then-else statement; hence, a secondary path of execution exists, followed when the condition evaluates false

3.3. Methodology for the Analysis of Suggested GPT Refactoring

To explore the capabilities of LLMs to refactor for-loops into Java streams, we used ChatGPT, as a representative of powerful LLMs, and more than two thousand inspected for-loops sampled from open source projects hosted on GitHub. Our study aims to provide answers to the following research questions.

- RQ_1 Can ChatGPT accurately refactor Java for-loops into stream pipelines while preserving the semantics of the generated code?
- RQ_2 Can ChatGPT overcome the limits of deterministic approaches based on pre-conditions to propose alternative for-loop refactoring?
- RQ_3 What limitations does ChatGPT encounter when proposing for-loop refactorings?

We answered such research questions by mining for-loops from several popular open-source Maven repositories. We mainly focused on repositories whose dependencies guaranteed ease of compiling the source code to make the validation of ChatGPT-generated solutions possible.

Figure 1 represents the three main steps of our methodology: the first one mines the for-loops candidates from the selected repositories; the second one asks ChatGPT to refactor the identified for-loops into stream-based pipelines; finally, the refactored code is injected in the source code and validated by compiling the application.

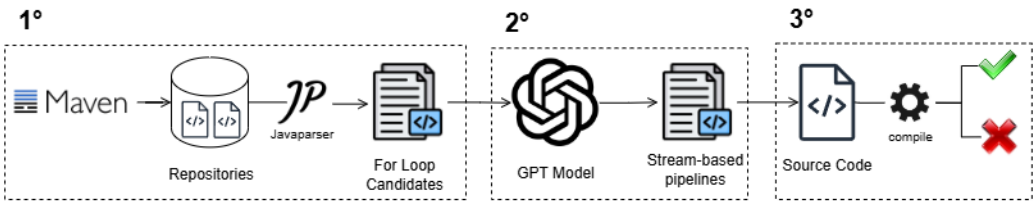

Figure 1. The methodology can be summarized in three main steps: mining for-loops, generating stream-based pipelines, and validation through compilation.

The objective of the first step was to systematically identify and extract for-loops by analyzing the source code of repositories [37]. By mining such repositories, we gathered valuable insights into how for-loops were implemented in real-world Java projects. We inspected the Maven repository [38] to identify the most popular projects suitable for our study. Starting from the highest-ranked projects, we applied two main selection criteria: (i) the presence of a substantial number of for-loops and (ii) the ability of the project to compile correctly without warnings or errors. We specifically focused on popular projects because their for-loops are likely written correctly and follow canonical coding practices, given the wide use and scrutiny of these codebases. Using these criteria, we selected five repositories that vary in size—both in terms of the number of files and lines of code—as well as in project scope. Our goal was to include a diverse set of projects representing different programming scenarios, ensuring that our analysis could be generalized across a range of real-world codebases.

To identify for-loops in the Java source files, the code was systematically parsed using the JavaParser library (https://javaparser.org, accessed on 23 May 2025), which allows source code analysis by means of an Abstract Syntax Tree (AST) [39], thereby providing a comprehensive representation of the source code's structure.

To perform code inspection, we employed the `VoidVisitorAdapter` class from JavaParser, which allows the definition of custom Visitor classes for traversing the abstract syntax tree (AST) of a Java source file and collecting specific nodes. Listing 3 shows the implementation of a `VoidVisitorAdapter` used to extract for-loops from the selected repositories. The custom visitor class, `CollectForEachStmt`, extends `VoidVisitorAdapter<HashSet <ForEachStmt»`, specifying that enhanced for-loop nodes (`ForEachStmt`) will be collected in a `HashSet`. The `visit()` method is overridden to define the behavior when a `ForEachStmt` node is encountered: it calls `super.visit()` to continue traversing child nodes and adds the visited node to the collection, ensuring that all loops are captured without duplication.

Method `extractForEachStmt()` takes a `CompilationUnit` as input, instantiates the custom visitor and a `HashSet` to store the collected loops, invokes the visitor on the compilation unit, and returns the set of extracted loops. Traditional for-loops (`ForStmt`) are handled similarly using a separate visitor class. By employing this approach, we ensure that all loops are extracted accurately and efficiently. The extracted for-loops are then saved into separate files, which are subsequently provided to ChatGPT for the automated refactoring analysis. This strategy allows the study to focus specifically on loop refactoring, leveraging both canonical and enhanced loop structures from real-world, widely used Java projects.

The next step involved refactoring such extracted loops into stream-based code using the GPT. This step was automated by leveraging the OpenAI API, hence interacting programmatically with the model. We used the `GPT-3.5 turbo` model to generate the refactoring. Table 3 reports the parameters used for GPT-based for-loop refactoring. These settings were chosen to mimic the behavior of a developer requesting the model to refactor a specific code fragment into a stream-based implementation, rather than providing the entire file or enabling random sampling. Listing 4 shows a Python snippet that interacts with the OpenAI GPT API to automatically refactor Java for-loops into stream-based code.

Function `ask_Gpt` constructs a system prompt, instructing the model to act as a skilled Java developer, and a user prompt containing the for-loop code to be refactored. It then sends these prompts to the GPT model via a chat completion request and returns the refactored code, ensuring that only the Java code is returned without explanations or additional formatting. Therefore, a prompt consists of two components: the system prompt and the user prompt. The system prompt defines the model's behavior through specific instructions, while the user prompt contains the input that describes the task. The formulation of these prompts significantly influences the LLM's code generation performance [40,41]. We designed our system and user prompts as shown in the following box.

Listing 3. A code fragment of the Visitors class used to identify and extract for-loop from the selected repositories.

```
public class Visitors {
....
    public static class CollectForEachStmt extends
        VoidVisitorAdapter<HashSet<ForEachStmt>>{
        @Override
        public void visit(ForEachStmt md, HashSet<ForEachStmt> methods) {
            super.visit(md, methods);
            methods.add(md);
        }
    }
    ....
    public static HashSet<ForEachStmt> extractForEachStmt(CompilationUnit
        cu){
        VoidVisitor<HashSet<ForEachStmt>> collectForEach = new
            CollectForEachStmt();
        HashSet<ForEachStmt> forEachstmts = new HashSet<>();
        collectForEach.visit(cu, forEachstmts);
        return forEachstmts;
    }
....
}
```

Listing 4. Example of the API call to the GPT model used to request the refactoring of a for-loop into a stream-based implementation.

```
from openai import OpenAI
from colorama import Fore, Style

client = OpenAI(api_key="")

def ask_Gpt(code):
    systemPrompt = "You are a software developer, skilled in writing Java
        source code and refactoring for loops into stream-based code"
    userPrompt = f"Please refactor this code to use streams, and provide
        only the Java code without any markdown, explanation, or comments:
        {code}"

    completion = client.chat.completions.create(
        model="gpt-3.5-turbo-0125",
        messages=[
            {"role": "system", "content":systemPrompt},
            {"role": "user", "content": userPrompt}
        ]
    )

    message = completion.choices[0].message.content
    print(f"{Fore.LIGHTBLUE_EX}{message}{Style.RESET_ALL}\n")
    return message
```

Table 3. Parameters used for GPT-based for-loop refactoring experiments.

Version	Temperature	Attempts	Context	Sampling
GPT-3.5 turbo	1.0	1	code fragment (not the entire file)	no

System and user prompts used for generation

System: 'You are a software developer, skilled in writing Java source code and refactoring for loops into stream-based code'
User: 'Please refactor this code to use streams, and provide only the Java code without any markdown, explanation, or comments: *code*'

Our prompt design follows the study proposed by Liu et al. [42] to ensure effective transformation of traditional for-loops into stream pipelines. To validate the clarity and correctness of the prompts, we manually executed them on representative code snippets, confirming that the model accurately understood the task and consistently generated the intended stream-based refactorings. By carefully structuring the prompts and verifying their effectiveness, we aimed to emulate the practical workflow of a developer using ChatGPT for targeted code refactoring.

To streamline the following refactoring process, we asked ChatGPT to omit any markdown, comment, or explanation, guaranteeing that the provided answer contained just the source code that will replace the original for-loop.

The last step aimed at injecting the stream-based code fragment into the proper method in the original source code. Each for-loop was replaced with the corresponding stream code. The injected code was then compiled using Maven. Each pair, for-loop, and corresponding stream-based code, was labeled according to the output of the compilation, which can be successful or not. If compilation was successful, refactoring was considered correct; otherwise, Maven output was collected to understand the issue.

4. Results of the Experiments

This section describes the results and addresses the three research questions expressed in Section 3.3. The study involved examining five GitHub projects, selected from the most recent Maven-based projects. Maven was chosen as the project manager due to its simplicity in compiling source code, ensuring semantic correctness. Since Java streams require Java 8 or higher, the focus was on recent repositories to avoid compatibility issues and ensure that the repositories are still actively maintained. The selected repositories are magpie [43], j2clmavenplugin [44], iridium [45], openrefine [46], and rapiddweller-benerator-ce [47]. Table 4 shows the number of stars, forks and contributors for each repository.

Table 4. GitHub characteristics of the selected repositories for the analysis

Repository	Stars	Forks	Contributors
iridium	123	17	11
magpie	191	33	15
j2clmavenplugin	8	3	3
rapiddweller-benerator-ce	151	26	10
openrefine	11,500	2100	395

4.1. RQ1: Semantic Preserving Transformation

The goal is to update the code to a more modern style while preserving the original semantics, ensuring that the new implementation does not encounter compilation problems, i.e., errors that were not present before.

Table 5 illustrates the results of the automated refactoring process carried out by ChatGPT on the five open-source repositories. The table columns display the repository name, the total number of for-loops identified, the number of successfully refactored loops, and the resulting status of the refactored code—specifically, whether it was compilable or uncompilable.

Table 5. Summary of for-loop refactoring outcomes across repositories.

Repository	For-Loops	Refactored	Compilable	Uncompilable
iridium	36	36	16	20
magpie	84	84	17	67
j2clmavenplugin	56	56	19	37
rapiddweller-benerator-ce	729	725	357	368
openrefine	1227	1200	196	1004
	2132	2101	605	1496

From the data, we see that in all cases, ChatGPT attempted to refactor the majority of for-loops in each repository. In the smallest repository, i.e., iridium, all 36 for-loops were refactored, and just 16 were compilable, leaving 20 uncompilable. Similar trends are seen in other repositories, such as magpie and j2clmavenplugin, where the majority of the for-loops were refactored; however, a significant portion of the resulting code was uncompilable. This points to the complexity involved in fully automating the refactoring process, especially when converting legacy for-loops to streams.

The larger repositories, rapiddweller-benerator-ce and openrefine, demonstrate the challenges of scaling the refactoring process. In rapiddweller-benerator-ce, 725 out of 729 loops were refactored by ChatGPT; however, only 357 were compilable, showing that larger codebases present more obstacles to successful refactoring. The largest repository, openrefine, had 1200 loops refactored, but only 196 were compilable, leaving over 1000 loops uncompilable.

In total, ChatGPT refactored 2101 out of 2132 identified for-loops, achieving a high success rate in terms of completing the refactoring itself. Only 605 of the refactored loops were compilable, while 1496 were uncompilable. This suggests that while ChatGPT is effective in automating loop refactoring, there are limitations to the process when it comes to ensuring that the resulting code is complete and readily compilable. The reasons for uncompilable code included contextual issues, syntax challenges, or differences in how stream-based code interacts with the rest of the codebase.

The inability of ChatGPT to refactor 31 for-loops is primarily influenced by the complex dependencies and potential side effects inherent to traditional loops. Streams, being designed with a functional and side-effect-free paradigm, contrast with the imperative style of for-loops. Additionally, streams are not always optimal for scenarios involving multi-level iteration, particularly when loops are interdependent or when there is intricate state sharing.

We manually evaluated a significant subset of the cases that successfully compiled. The manually inspected snippets were around 25% of the ones that compiled. We found that the refactored code preserved the original semantics; hence, the transformations were correct, most of the time, with only around 10% of code not being semantically correct. For the cases that failed to compile, we analyzed the code to identify the causes of compilation errors, categorizing them according to the type of issue encountered. The classification, along with examples and detailed discussion, is presented in Section 4.3. This process allowed us to rigorously assess both the correctness of successful refactoring and the limitations of the automatic approach in handling more complex or unconventional code patterns.

Gathered data underscore the potential of AI-driven code refactoring tools and highlight the need for human intervention to review and fine-tune the results, especially in

complex or large-scale projects. ChatGPT is a helpful tool for programmers; however, it is not yet reliable enough to be used without human oversight [8].

> Answer to RQ1. Only 29% of the automatically refactored for-loops compiled successfully, and a manual evaluation suggested that these preserved the original semantics. The remaining 71% failed to compile due to (sometimes minor) flaws, underscoring the need for human review and refinement of the refactoring results produced by ChatGPT.

4.2. RQ2: Comparison with SOTA Approaches

State-of-the-art approaches for automated for-loop refactoring typically impose strict preconditions on the loops they can transform, limiting their applicability to only a subset of 'simple' or canonical loops. These preconditions often exclude loops with multiple return statements, break statements, local variables referenced from outside the loop, or throwing clauses, under the assumption that such constructs are too complex for reliable transformation. With this research question, we aim to challenge this perspective by showing that the GPT is capable of refactoring many loops that do not satisfy these traditional preconditions. Our results highlight that these 'old' limitations are overly conservative and that LLM-based approaches can handle a wider range of for-loop patterns, generating compilable stream-based code in scenarios that previous SOTA techniques would have deemed ineligible.

Tables 6–8 show the number of for-loops generated by ChatGPT and classified according to the categories discussed in Section 3.2. Each column is a category, and the two sub-columns 'c' and 'u' give the number of compilable and uncompilable for-loops generated, respectively.

Table 6. Number of for-loops that do not satisfy the first four preconditions listed in Table 1 and that were refactored by ChatGPT successfully (column c) or not (column u). Columns 'c' and 'u' indicate compilable and uncompilable, respectively.

Repository	LV		TC		BR		MR	
	c	u	c	u	c	u	c	u
iridium	0	1	4	0	1	6	0	0
magpie	0	0	2	5	0	1	0	0
j2clmavenplugin	0	0	2	4	0	0	0	0
rapiddweller-benerator-ce	11	22	12	45	11	8	0	5
openrefine	31	43	10	59	27	28	1	2

Table 7. Number of for-loops that do not satisfy the last four preconditions in Table 1, and were refactored successfully by ChatGPT (column c) or unsuccessfully (column u). Columns 'c' and 'u' indicate 'compilable' and 'uncompilable', respectively.

Repository	CO		MS		NL		SW	
	c	u	c	u	c	u	c	u
iridium	0	0	7	2	2	2	2	0
magpie	0	0	4	10	1	1	0	0
j2clmavenplugin	2	7	6	12	2	5	0	0
rapiddweller-benerator-ce	3	8	70	124	14	32	5	1
openrefine	32	48	91	351	59	68	2	2

Table 8. Number of for-loops refactored by ChatGPT for the templates presented in [12] (see Table 2). Columns 'c' and 'u' indicate compilable and uncompilable, respectively.

Repository	OR		SS		1st		2nd		3rd		4th		5th	
	c	u	c	u	c	u	c	u	c	u	c	u	c	u
iridium	0	1	9	18	0	1	0	0	0	0	0	4	0	0
magpie	1	3	13	57	0	0	0	0	0	0	4	36	0	0
j2clmavenplugin	1	0	13	25	0	0	0	0	0	0	0	0	0	0
rapiddweller-benerator-ce	9	37	287	244	4	18	2	6	3	4	34	10	3	0
openrefine	31	54	105	653	19	33	0	1	14	10	31	12	4	3

Table 6 shows the number of for-loops that were classified according to their characteristics, i.e., having local variables (LV), having throwing clauses (TC), having break statements (BR), and having more than one return (MR); however, they do not satisfy the respective preconditions presented in Table 1. Despite this, ChatGPT proposed a corresponding stream-based code fragment, whose successful compilation showed that its refactoring was possible. Although some transformations were successful, others posed significant challenges. A loop that does not satisfy the Throw Clause (TC) precondition is less likely to have a corresponding successful stream-based code. Specifically, in the rapiddweller and openrefine repositories, fewer than 22% of the generated fragments were compilable. Similarly, loops not satisfying the more return (MR) precondition resulted in very few compilable generated fragments for both repositories: 100% and 50%, respectively. These findings suggest that certain code blocks, i.e., those involving throwing exceptions and having many return statements, are more complex or less suitable for automated refactoring into stream pipelines since their paradigms are more relatable to an imperative style of programming rather than a functional one.

The largest number of refactoring attempts, as well as the highest rate of successful refactorings, occurred in the openrefine and rapiddweller-benerator-ce repositories. In openrefine, 31 loops not satisfying the precondition local variables (LV) were refactored successfully; however, 43 failed to compile. This indicates a substantial attempt to convert complex for-loops that involved more than one reference to non-effectively final variables defined outside the loop, albeit with a significant failure rate. Similarly, this repository saw notable success in refactoring loops not satisfying the BR precondition, with 27 successful compilations, although again accompanied by a comparable number of failures (28).

In contrast, smaller repositories, such as iridium, magpie, and j2clmavenplugin, saw relatively few refactoring attempts, with mixed results. In iridium, e.g., the Throw Clause (TC) precondition resulted in a full success rate with four compilable loops and no failures. This indicates that simpler loop structures, may be more amenable to automated refactoring despite containing exception-handling statements.

The refactoring of loops when the Break (BR) precondition was not satisfied consistently showed a higher success rate across the repositories, particularly in rapiddweller and openrefine, where the majority of transformed code snippets compiled successfully. This suggests that loops containing break statements can be effectively transformed into streams, even though state-of-the-art approaches have claimed that such cases are not feasible (we could more appropriately say not straightforward) for stream refactoring.

The overall success rates of compilable refactorings vary widely between repositories and preconditions, with certain combinations that prove more conducive to stream-based transformations. Notably, the openrefine repository shows both the highest volume of attempted refactorings and a reasonable number of successful compilations across multiple preconditions, indicating that it contains a broad variety of loops that are somewhat compatible with Java's Stream API. However, the relatively high number of failures in all preconditions indicates that while ChatGPT can handle simpler cases, more intricate loop structures involving complex control flow or variable manipulation remain challenging.

Listing 5 shows an example of a for-loop that does not satisfy the precondition local variables (LV); however, it was successfully refactored by ChatGPT to an equivalent stream pipeline. In the original loop, the local variable columnIndex is computed within the loop and used in the conditional check and to compute the local variable cellIndex, which is then assigned to an array. In the refactored version, these local variables are handled implicitly: the stream maps the loop index i to columnIndex, filters according to the same logic, maps to cellIndex, and finally performs the side-effect update on keyedGroup.cellIndices. This transformation demonstrates that ChatGPT can correctly manage multiple intermediate variables by embedding their computation within the stream pipeline stages.

Listing 5. Example of for-loop refactoring not satisfying the local variables (LV) precondition as it contains more than one local variable and successfully refactored by ChatGPT.

```
// original for loop
for (int i = 0; i < group.columnSpan; i++) {
    int columnIndex = group.startColumnIndex + i;
    if (columnIndex != group.keyColumnIndex && columnIndex <
        columnModel.columns.size()) {
        int cellIndex = columnModel.columns.get(columnIndex).getCellIndex();
        keyedGroup.cellIndices[c++] = cellIndex;
    }
}

// refactored stream pipeline
IntStream.range(0, group.columnSpan)
    .mapToObj(i -> group.startColumnIndex + i)
    .filter(columnIndex -> columnIndex != group.keyColumnIndex &&
        columnIndex < columnModel.columns.size())
    .map(columnIndex -> columnModel.columns.get(columnIndex).getCellIndex())
    .forEach(cellIndex -> keyedGroup.cellIndices[c++] = cellIndex);
```

Table 7 shows the results of the refactoring transformations attempted when considering the loops that exhibit the characteristics shown in Table 1 (max statements, MS; nested loops, NL; switch statements, SW; and continue statement, CO); however, such loops do not satisfy the respective preconditions given.

Transforming loops is less feasible when the loop body has more than five statements (hence they are classified as max statements (MS) loops). The results show that for repositories such as rapiddweller-benerator-ce and openrefine, ChatGPT struggles with this constraint, with 124 and 351 uncompilable cases, respectively. In contrast, iridium has a relatively balanced ratio of compilable (7) to uncompilable (2) cases. For repositories magpie and j2clmavenplugin, a higher number of uncompilable cases were given by ChatGPT, indicating that many loops exceed the complexity threshold for stream-based refactoring.

Nested loops are notoriously difficult to refactor using streams as they introduce more intricate control flows. This is reflected in the high number of uncompilable cases across the board. Repositories openrefine and rapiddweller-benerator-ce are particularly affected, with 68 and 32 uncompilable cases, respectively. Only iridium and magpie manage a more balanced ratio of compilable to uncompilable cases, but still, the presence of nested loops is a substantial barrier to stream refactoring.

Switch statements pose another challenge for stream refactoring. However, the number of switch statements in loops is relatively small compared to other constraints. Repositories rapiddweller-benerator-ce and openrefine have a few compilable and uncompilable cases, but the impact is less severe overall. This suggests that switch statements are less common within loops in the analyzed repositories, or developers might be avoiding their usage in situations conducive to refactoring.

In the repositories, the number of statements in the loop (MS) and the presence of nested loops (NL) are the most significant barriers to refactoring. Particularly, openrefine

and rapiddweller-benerator-ce repositories show that a large portion of their loops exceed the complexity threshold, indicating that functional programming styles may not be suitable for these parts of the codebase. The results also show that control flow constructs such as switch statements (SW) are relatively rare in compilable stream refactoring.

Regarding for-loops including continue statements (CO), only j2clmavenplugin, rapiddweller-benerator-ce, and openrefine repositories have a few compilable cases, with openrefine standing out with 32 compilable cases. However, the large number of uncompilable cases in openrefine (48) and other repositories highlights that this remains a significant limitation.

While some repositories, such as iridium, have a more balanced number of compilable and uncompilable cases across multiple categories, repositories like openrefine and rapiddweller-benerator-ce present challenges in refactoring, primarily due to the complexity of loop bodies and control flow structures. These results suggest that while streams offer a more readable and functional style of programming, they are not always suitable for all types of loops, especially those with complex control flows or extensive nested structures.

Listing 6 shows an example of a for-loop that does not satisfy the precondition continue (CO); however, it was successfully refactored by ChatGPT to an equivalent stream pipeline. The original loop skips empty strings using a continue statement and applies a sequence of transformations to build a normalizedLocalName String. In the refactored version, the same logic was implemented using a stream pipeline: the filter step excludes empty strings, followed by two map operations to transform each path, and a reduce operation to concatenate the results. Despite the presence of the continue statement, which usually complicates refactoring, the transformation preserves the original behavior and produces an equivalent result using functional constructs.

Listing 6. An example of for-loop refactoring of the CO category where ChatGPT successfully refactors a for-loop containing a continue statement.

```
// original for loop
for (String p : paths) {
    if (p.equals("")) {
        continue;
    }
    p = currentFileSystem.toLegalFileName(p, '-');
    normalizedLocalName += String.format("%c%s", File.separatorChar, p);
}

// refactored stream pipeline
normalizedLocalName = paths.stream()
    .filter(p -> !p.equals(""))
    .map(p -> currentFileSystem.toLegalFileName(p, '-'))
    .map(p -> String.format("%c%s", File.separatorChar, p))
    .reduce("", (result, p) -> result + p);
```

Table 8 presents the evaluation of loops transformed into Java stream-based code for the cases of loops with one return statement (OR), loops having fewer than five statements, and the five templates presented in [12]. The results reveal variability in both the success and failure rates of stream-based refactoring depending on the repository and the type of for-loop template applied. Notably, the counts of compilable versus uncompilable transformations indicate that certain patterns of loops are more challenging to refactor into streams, particularly with template 1 (for-loops with a conditional statement and one or more return statements), in agreement with the results of Table 6 where statements with returns are more challenging.

Starting with loops having satisfied the one return (OR) precondition resulted in very few compilable generated fragments for both repositories: less than 20% and 37%, respectively. In contrast, column small size (SS) represents cases where the loop body size is less than five statements. Streams are generally suitable for such loops, and we

observe a higher number of compilable cases in most repositories. For instance, openrefine has 105 compilable cases, but it is also noteworthy that there are 653 uncompilable cases, emphasizing that many loops are still too complex for straightforward refactoring. A deeper analysis of the compilation error is described in Section 4.3 to elucidate the main reasons behind the large number of uncompilable cases. Repository rapiddweller-benerator-ce shows a large number of cases (287 compilable vs. 244 uncompilable), suggesting a mixed suitability for streams in this repository.

The first template was found in two repositories: rapiddweller-benerator-ce and openrefine. Interestingly, openrefine achieved 19 compilable refactorings, but 33 instances failed to compile. This suggests that the first template, while somewhat effective, still poses significant challenges, particularly due to conditional structures that may not easily translate into a single stream operation. Similarly, rapiddweller-benerator-ce encountered several unsuccessful refactoring, with just 4 compilable for-loops and 18 uncompilable ones.

The second and fifth templates show few occurrences, showing that they are less common when writing for-loop structures. Nevertheless, the fifth template shows a higher success rate than the second, and this highlights that the temp variable's inclusion complicates the transformation, possibly due to scoping and reuse issues as streams are typically expected to be stateless and work more effectively with immutable data.

The results for the third template vary, and it appears to be another challenging pattern, particularly in openrefine (14 compilable and 10 uncompilable) and rapiddweller-benerator-ce (3 compilable and 4 uncompilable). The combination of a temp variable and method calls adds complexity that might interfere with stream semantics, particularly since method calls inside loops may cause side effects, which are difficult to express in functional streams.

The forth template is the most prevalent in the data, particularly in magpie (4 compilable and 36 uncompilable), rapiddweller-benerator-ce (34 compilable and 10 uncompilable), and openrefine (31 compilable and 12 uncompilable). Despite a high number of successful compilations in some repositories, the number of uncompilable transformations remains significant. This suggests that while this template fits well within the stream paradigm (as streams naturally support element-wise operations), edge cases, such as incorrect handling of the new value insertion logic, might still cause failure in the refactoring process.

The data suggest that stream-based refactoring is highly dependent on the specific structure of the loop and the repository's coding patterns. While some templates (like the fourth) are frequently encountered and refactored with relative success, others (such as the second and third) are more error-prone. This variability highlights the need for further improvements in automated refactoring tools to handle edge cases, particularly where temp variables and conditional branches are involved. Additionally, individual codebases vary significantly in how amenable they are to stream-based refactoring, suggesting that the nature of the existing code plays a crucial role in determining the success rate of such transformations.

According to the results previously shown, ChatGPT was able to correctly generate a refactored version of 414 for-loops that were originally discarded by the SOTA approaches because they violated their preconditions. This highlights that ChatGPT has overcome the limits of these approaches, suggesting innovative refactoring that was not previously proposed. This can be justified by the massive training process of the GPT model, which it applied to the billions of model code snippets that, unlike deterministic approaches, helped the model to properly combine stream APIs to obtain innovative stream pipelines. The remaining 903 uncompilable generated for-loops can be further analyzed to comprehend the reasons behind these failures.

Listing 7 displays an example of a for-loop corresponding to the fifth template presented in [12] that was successfully refactored by ChatGPT. The original code iterates over

the list sg.getSnaks() using a classic enhanced for-loop. Inside the loop, it conditionally selects between two visitors—referenceSnakPrinter or mainSnakPrinter—depending on the value of the boolean variable reference and then writes the result of the accept method to the writer. The refactored version expresses the same logic using a stream pipeline. It transforms the collection into a stream, applies a map operation that performs the same conditional logic inline using the ternary operator, and finally consumes the result using forEach(writer::write).

Listing 7. An example of for-loop refactoring of the 5th template where ChatGPT successfully refactored a loop containing an if/else statement.

```
// original for loop
for (Snak s : sg.getSnaks()) {
    if (reference) {
        writer.write(s.accept(referenceSnakPrinter));
    } else {
        writer.write(s.accept(mainSnakPrinter));
    }
}

// refactored stream pipeline
sg.getSnaks().stream()
    .map(s -> reference ? s.accept(referenceSnakPrinter) :
        s.accept(mainSnakPrinter))
    .forEach(writer::write);
```

Answer to RQ2. ChatGPT successfully generated 414 innovative stream pipelines, surpassing the limitations of previous state-of-the-art (SOTA) approaches. The proposed refactoring effectively addresses the preconditions set by these approaches, offering interesting and novel solutions.

4.3. RQ3: Compilation Error Analysis

To address the third research question, we focus the study on the uncompilable refactorings generated by ChatGPT (see Section 4.2). Table 9 highlights the metrics extracted from the compilation errors. The columns are described according the tags introduced in Section 3.2, while rows represent the error name collected from the compilation report. A description of each error can be found in Table A1 in the Appendix A.

From Table 9, the 'cannot find symbol' error is the most prevalent across all categories. This suggests that these categories often involve references to variables or methods that are not defined within the scope. This could be due to misspelled variable or method names, variables or methods that are declared outside the loop but not accessible within it, and missing imports or incorrect package references.

The 'illegal character' error is relatively rare, appearing only in SS (seven occurrences) and the fifth template (six occurrences). Illegal characters can be caused by typographical error, copy-pasting code from different sources that include non-printable characters, and incorrect encoding settings in the development environment.

Table 9. Numbers of all compilation errors encountered for each analyzed category

Error	LV	TC	BR	OR	MR	CO	MS	SS	NL	SW	1st	2nd	3rd	4th	5th
illegal character	0	0	0	0	0	0	0	7	0	0	0	0	0	6	0
cannot find symbol	55	84	33	66	6	48	361	451	88	2	31	7	9	26	1
symbol expected	0	2	0	0	0	1	11	59	3	0	0	0	3	37	0
unreported exception	1	6	0	0	0	5	15	56	5	0	0	0	0	0	0
local variables referenced	11	7	7	1	0	10	34	18	5	1	0	0	0	0	0
variable is already defined	13	4	11	4	0	5	12	57	3	0	2	0	0	2	0
incompatible types	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0
non-static variable	0	0	0	4	0	0	1	3	0	0	1	2	0	0	0
no suitable method	0	0	1	3	0	0	1	5	1	0	2	0	0	0	0
illegal start of type	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
try without	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
parse errors	4	2	2	2	0	4	14	21	11	0	1	0	2	0	1

The ‘symbol expected’ error is notable in SS (59 occurrences) and the fifth template (37 occurrences). It indicates syntax issues where the compiler expected a different token. Common causes include missing semicolons or braces, incorrect use of operators or delimiters, and incomplete statements or expressions.

Another common error is ‘unreported exception’, which occurs in SS (56 occurrences) and MS (15 occurrences). It suggests that these categories often involve operations that can throw exceptions not handled properly. This can happen when methods that throw checked exceptions are called without proper try–catch blocks and the throws clause is missing in the method signature.

‘Local variables referenced’ is often overlooked during refactoring. This error is relatively frequent in MS (34 occurrences) and SS (18 occurrences). It indicates issues with variable scope and usage, such as referencing variables outside their declared scope, modifying effectively final variables within lambda expressions, and conflicts between local and global variable names.

The ‘variable is already defined’ error is notable in SS (57 occurrences) and LV (13 occurrences). It suggests multiple declarations of the same variable within these categories. This can occur due to re-declaring variables within nested scopes, conflicts between parameter names and local variable names, and copy-pasting code without renaming variables.

The ‘incompatible types’ error is rare, with only four occurrences in SS. It indicates type mismatch issues, which can be caused by assigning values of incompatible types, incorrect method return types, and using raw types instead of parameterized types in generics.

The ‘non-static variable’ error appears in OR (four occurrences) and SS (three occurrences). It indicates issues with accessing non-static variables from static contexts. This can happen when trying to access instance variables from static methods, when the scope of static and non-static members is misunderstood, and when the static context is used incorrectly within lambda expressions.

The ‘no suitable method’ error is relatively rare, with the highest occurrence in SS (five occurrences). It indicates method signature mismatches, which can be caused by calling methods with incorrect parameters, overloading methods without proper parameter types, and using incorrect method references in lambda expressions.

The ‘illegal start of type’ error is very rare, with only one occurrence in SS. It indicates a syntax issue at the start of a type declaration, which can be caused by incorrectly placed annotations or modifiers, missing class or interface keywords, and typographical errors in type declarations.

The ‘try without’ error is extremely rare, with only one occurrence in SS. It indicates a missing try block, which can happen when forgetting to include the try keyword before a block of code that handles exceptions, incorrectly formatting the try–catch–finally structure, and overlooking exception handling during refactoring.

Finally, 'parse errors' is more common in SS (21 occurrences) and NL (11 occurrences). It indicates general syntax errors, which can be caused by incomplete or incorrect code statements, misplaced or missing punctuation, and errors in code structure or formatting.

The types of errors encountered can provide insights into both the severity of mistakes made by ChatGPT and potential underlying causes. Moreover, they offer an indication of the effort a developer may need to invest to resolve them.

Starting with the most common issue, the 'cannot find symbol' error often arises due to insufficient context provided to ChatGPT during code generation. Since the model was queried with a focus solely on refactoring a for-loop (see Section 3.3), it generated the corresponding stream-based code without including necessary imports and variable declarations, assuming that they are implicit. In these cases, the error can easily be rectified by adding the required imports or defining the missing variables. However, a more complex situation occurs when the missing symbol refers to a class or method that does not exist, having been 'hallucinated' by the model [48,49]. This could require more significant intervention by the developer.

Similarly, the 'variable is already defined' error is a result of inadequate context. In this case, the model unnecessarily redefines variables already declared outside the generated stream pipeline. The resolution here is straightforward: remove the redundant variable declaration and use the pre-existing one.

Errors such as 'illegal character', 'symbol expected', 'illegal start of type', and 'parse errors' fall under the category of syntactical mistakes. These errors result from the model improperly writing code, using incorrect symbols, or omitting required ones. These errors are typically simple to correct, requiring the developer to insert or replace the appropriate symbols.

Shifting to more complex errors, 'unreported exception' and 'try without' relate to exception handling in Java, specifically within the context of stream APIs. These errors occur when the model fails to properly handle exceptions, a task often better suited for traditional for-loops rather than stream pipelines. Fixing such errors may necessitate reverting to a more conventional approach or a detailed restructuring of the code to ensure proper exception handling, which diminishes the utility of stream-based refactoring.

Although rarer, the 'incompatible types' and 'non-static variable' errors demand a deeper contextual analysis. These errors typically occur when there is a mismatch between the expected and actual types or when non-static variables are used incorrectly within static contexts. Resolving these issues may not always be feasible due to the constraints imposed by the semantic structure of stream pipelines.

Finally, the 'local variable referenced from a lambda expression must be final or effectively final' error is tightly connected to one of the preconditions discussed in state-of-the-art approaches (see Section 4.2). By design, stream APIs do not allow the use of non-final local variables, meaning that certain for-loops cannot be easily refactored into streams. However, as indicated in Table 6, ChatGPT is often able to successfully refactor these loops despite this limitation, highlighting its potential to handle such complex refactorings.

Listing 8 highlights a typical compilation error that can arise when refactoring a for-loop into a Java stream. The generated code attempts to reassign the local variable `fileName` inside a lambda expression used within the `ifPresent` method. However, Java requires that variables captured by lambdas be final or effectively final, and reassignment violates this rule. As shown in the Maven output, the compiler fails with the error 'local variables referenced from a lambda expression must be final or effectively final', preventing successful compilation of the refactored code. Such variable scope and mutability dependencies are often subtle and non-trivial, making them particularly difficult for LLMs to detect and handle correctly during automated refactoring.

Listing 8. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `local variables referenced`.

```
Arrays.stream(new String[] { ".gz", ".bz2" })
    .filter(ext -> fileName.endsWith(ext))
    .findFirst()
    .ifPresent(ext -> fileName = fileName.substring(0,
        fileName.length() - ext.length()));

....
[ERROR] Failed to execute goal
    org.apache.maven.plugins:maven-compiler-plugin:3.13.0:compile
    (default-compile) on project main: Compilation failure: Compilation
    failure:
[ERROR]
    /C:/Users/Ale-m/Desktop/RepositoryMining/AnalyzedRepositories/openrefine
    /main/src/com/google/refine/importing/ImportingUtilities.java:
    [733,24] local variables referenced from a lambda expression must be
        final or effectively final
....
```

Answer to RQ3. ChatGPT successfully generated a range of stream pipelines but encountered errors primarily due to the lack of context during code generation. Common issues like 'cannot find symbol' were often easily resolved by adding imports or defining variables, while more complex errors such as handling exceptions in streams required deeper analysis.

5. Threats to Validity

This section discusses potential threats to the validity of our study, highlighting factors that may affect the reliability, interpretability, and generalizability of our findings. We categorize these threats into internal and external validity. Internal validity addresses the fidelity of the study's execution, including methodological choices, tool behavior, and dataset characteristics that could influence the observed outcomes. External validity concerns the extent to which our results can be generalized to other codebases, loop patterns, or large language models beyond the specific configurations and repositories considered in this work. By explicitly acknowledging these limitations, we aim to provide a transparent assessment of the study's strengths and constraints.

Internal validity pertains to the integrity of the study's execution. The non-deterministic nature of GPT-3.5, combined with the choice of hyperparameters and prompts, may influence the quality of the generated content without guaranteeing optimal solutions. Furthermore, since GPT-3.5's training data is not publicly available, there is a possibility that similar code patterns exist in its training set, potentially providing an advantage over benchmark solutions. We used the default parameters of GPT-3.5 (temperature = 1.0), which could maximize response diversity. This choice reflects typical developer usage as ChatGPT's UI does not provide options to adjust the temperature. Keeping the default parameters ensures that the experimental scenario closely resembles real-world usage. Finally, we selected five repositories from the Maven Central Repository. While this selection may not represent all publicly available projects, we aimed to choose the most popular ones, reflecting those most widely used by developers. Additionally, we ensured diversity in project type, scope, and size, in order to capture a range of realistic programming scenarios.

External validity concerns the generalizability of our findings. While our study focused on fifteen selected for-loop idioms, this set does not encompass all possible refactoring proposals, potentially limiting generalizability. However, prior studies emphasize these idioms as among the most widely used and impactful in Java development, supporting their relevance. Additionally, our analysis involves methods extracted from diverse GitHub

repositories, highlighting the practical need to address missing refactorings. The number of extracted loops and the popularity of the chosen repositories should represent the majority of for-loop idioms. Moreover, the high number of compilation errors in some categories suggests that refactoring difficulties arise, especially when the code references or defines local variables or uses a context that is beyond the loop itself. Such recurrent difficulties should represent a common occurrence in for-loops. The study focuses exclusively on ChatGPT, without considering other LLMs. Nevertheless, ChatGPT is widely regarded as a state-of-the-art LLM, serving as a representative model. Future research could enhance generalizability by evaluating ChatGPT alongside other LLMs across a broader range of refactorings.

6. Conclusions

The adoption of the functional programming paradigm introduced in Java 8 offers significant advantages in terms of code conciseness, expressiveness, and abstraction. Despite these benefits, the transition from imperative constructs, such as for-loops, to stream pipelines remains underutilized in practice. This paper presented an empirical study evaluating ChatGPT's effectiveness in refactoring for-loops into stream pipelines compared to state-of-the-art (SOTA) approaches. Our findings indicate that ChatGPT can successfully refactor a substantial portion of loops while preserving code semantics, demonstrating its potential as a valuable tool for code transformation.

However, our analysis also revealed that ChatGPT's performance is inconsistent across different loop categories. Although it excels in cases that align closely with SOTA templates, it struggles with more complex loops that require nuanced contextual understanding or violate the preconditions of existing deterministic approaches. Notably, ChatGPT produced correct refactorings beyond the limitations of SOTA in specific cases, suggesting its capacity to generalize transformations where rigid rule-based methods fall short. Conversely, we identified failure patterns where ChatGPT either generated syntactically invalid code or produced semantically incorrect transformations, underscoring the need for human oversight.

These findings suggest a complementary role for ChatGPT alongside deterministic approaches. By leveraging ChatGPT's ability to generalize for diverse patterns and combining it with the precision of formal refactoring rules, future research could explore hybrid solutions that improve automation and reliability. Further investigations should also examine ways to mitigate failure cases and improve the interpretability of ChatGPT's outputs. This study provides a foundation for the advancement of automated refactoring techniques and underscores the potential of large language models to bridge the gaps left by traditional methodologies.

Author Contributions: Conceptualization, A.M. and E.T.; methodology, A.M.; software, A.M.; validation, A.M. and E.T.; data curation, A.M.; writing—original draft preparation, A.M.; writing—review and editing, E.T.; supervision, E.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data derived from public domain resources.

Acknowledgments: We acknowledge the support of the University of Catania PIACERI project TEAMS.

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

Appendix A

Table A1. Compilation error types with descriptions.

Error	Description
illegal character	The code contains a character not allowed in Java source code, often due to copy-paste or encoding issues.
cannot find symbol	A referenced variable, method, or class is undefined or out of scope.
symbol expected	Java expected a symbol (e.g., semicolon, bracket, or identifier) but found none or something invalid.
unreported exception	A checked exception may be thrown but is not caught or declared in the method signature.
local variables referenced	A local variable is used before it has been initialized.
variable is already defined	A variable is declared more than once in the same scope.
incompatible types	An assignment or operation involves mismatched data types that cannot be converted automatically.
non-static variable	A non-static variable is referenced from a static context.
no suitable method	A method call does not match any method signature available for the object.
illegal start of type	A statement or declaration is placed incorrectly in the class or method structure.
try without	A `try` block is missing a corresponding `catch` or `finally` block.
parse errors	Generic syntax errors preventing the compiler from parsing the code correctly.

Listing A1. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `cannot find symbol`.

```
1  XMLUtil.getChildElements(element)
2      .stream()
3      .filter(child -> !MEMBER_ELEMENTS.contains(child.getNodeName()))
4      .map(child -> result.addSubStatement(context.parseChildElement(child,
           parentXmlPath, currentPath)))
5      .collect(Collectors.toList());
6
7  ....
8  [ERROR] Failed to execute goal
       org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile
       (default-compile) on project rapiddweller-benerator-ce: Compilation
       failure: Compilation failure:
9  [ERROR] ... CascadeParser.java:[70,5] cannot find symbol
10 [ERROR]   symbol:   method stream()
11 [ERROR]   location: class org.w3c.dom.Element[]
12 [ERROR] ... CascadeParser.java:[73,14] cannot find symbol
13 [ERROR]   symbol:   variable Collectors
14 [ERROR]   location: class
       com.rapiddweller.benerator.engine.parser.xml.CascadeParser
15 ....
```

Listing A2. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `symbol expected`.

```
values.entrySet().stream()
    .forEach(entry -> MAP.put(entry.getKey(), new
        IncrementalIdGenerator(Long.parseLong(entry.getValue()))));

....
[ERROR] Failed to execute goal
    org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile
    (default-compile) on project rapiddweller-benerator-ce: Compilation
    failure
[ERROR] ... LocalSequenceGenerator.java:[136,107] ')' expected
....
```

Listing A3. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `unreported exception`.

```
applicationTypes.stream()
    .filter(typeEntity ->
        typeEntity.getType().equals(ApplicationType.SINGLE_PAGE))
    .map(typeEntity ->
        ApplicationGenerator.generateIridiumApplication(entityManager,
        iridiumTenant, typeEntity))
    .forEach(application -> this.iridiumAppId = application.getClientId());

....
[ERROR] Failed to execute goal
    org.apache.maven.plugins:maven-compiler-plugin:3.11.0:compile
    (default-compile) on project iridium-cli: Compilation failure
[ERROR] ... InitCommand.java:[42,71] unreported exception
    java.security.NoSuchAlgorithmException; must be caught or declared to be
    thrown
....
```

Listing A4. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `variable already defined`.

```
String errorMessage = messages.stream()
    .collect(Collectors.joining("\n"));

....
[ERROR] Failed to execute goal
    org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile
    (default-compile) on project rapiddweller-benerator-ce: Compilation
    failure: Compilation failure:
[ERROR] ... CreateProjectPanel.java:[275,16] variable errorMessage is
    already defined in method showErrors(java.lang.Object...)
....
```

Listing A5. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `incompatible types`.

```
sources.stream()
        .filter(source -> !source.isThreadSafe())
        .findAny()
        .ifPresent(s -> { return false; });

....
[ERROR] Failed to execute goal
    org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile
    (default-compile) on project rapiddweller-benerator-ce: Compilation
    failure
[ERROR] ... MultiGeneratorWrapper.java:[149,19] incompatible types:
    unexpected return value
....
```

Listing A6. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `no suitable method`.

```
return Arrays.stream(unluckyNumbers)
    .anyMatch(test -> candidate.endsWith(test));

....
[ERROR] Failed to execute goal
    org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile
    (default-compile) on project rapiddweller-benerator-ce: Compilation
    failure
[ERROR] ... UnluckyNumberValidator.java:[189,22] no suitable method found
    for stream(java.util.Set<java.lang.String>)
....
```

Listing A7. Example of a compilation error encountered during the analysis of a refactored for-loop. The code snippet shows the stream generated by the GPT, followed by the Maven compilation failure indicating an error of type `illegal start of type`.

```
writeColumnInfos.stream()
        .map(info -> {
            Object jdbcValue = entity.getComponent(info.name);
            if (info.type != null) {
                jdbcValue = AnyConverter.convert(jdbcValue, info.type);
            }
            return new AbstractMap.SimpleEntry<>(info, jdbcValue);
        })
        .forEach(entry -> handleOracleType(tableName, statement,
            writeColumnInfos.indexOf(entry.getKey()), entry.getKey(),
            entry.getValue()););

....
[ERROR] Failed to execute goal
    org.apache.maven.plugins:maven-compiler-plugin:3.10.1:compile
    (default-compile) on project rapiddweller-benerator-ce: Compilation
    failure: Compilation failure:
[ERROR] ... AbstractDBSystem.java:[697,142] illegal start of expression
....
```

References

1. Urma, R.G.; Fusco, M.; Mycroft, A. *Modern Java in Action: Lambdas, Streams, Functional and Reactive Programming*, 2nd ed.; Manning Publications Co.: Shelter Island, NY, USA, 2018.
2. Mazinanian, D.; Ketkar, A.; Tsantalis, N.; Dig, D. Understanding the use of lambda expressions in Java. *Proc. ACM Program. Lang.* **2017**, *1*, 1–31. [CrossRef]
3. Møller, A.; Veileborg, O.H. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* **2020**, *4*, 1–29. [CrossRef]
4. Rosales, E.; Rosà, A.; Basso, M.; Villazón, A.; Orellana, A.; Zenteno, A.; Rivero, J.; Binder, W. Characterizing Java Streams in the Wild. In Proceedings of the 2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS), Hiroshima, Japan, 26–30 March 2022; pp. 143–152. [CrossRef]
5. Khatchadourian, R.; Tang, Y.; Bagherzadeh, M.; Ray, B. An Empirical Study on the Use and Misuse of Java 8 Streams. In *Fundamental Approaches to Software Engineering*; Wehrheim, H., Cabot, J., Eds.; Springer: Cham, Switzerland, 2020; pp. 97–118.
6. Nostas, J.; Alcocer, J.P.S.; Costa, D.E.; Bergel, A. How Do Developers Use the Java Stream API? In Proceedings of the Computational Science and Its Applications—ICCSA 2021, Cagliari, Italy, 13–16 September 2021; Springer: Cham, Switzerland, 2021; pp. 323–335.
7. Champa, A.I.; Rabbi, M.F.; Nachuma, C.; Zibran, M.F. ChatGPT in Action: Analyzing Its Use in Software Development. In Proceedings of the MSR '24: 21st International Conference on Mining Software Repositories, Lisbon, Portugal, 15–16 April 2024; pp. 182–186. [CrossRef]
8. DePalma, K.; Miminoshvili, I.; Henselder, C.; Moss, K.; AlOmar, E.A. Exploring ChatGPT's code refactoring capabilities: An empirical study. *Expert Syst. Appl.* **2024**, *249*, 123602. [CrossRef]
9. Bulla, L.; Midolo, A.; Mongiovì, M.; Tramontana, E. EX-CODE: A Robust and Explainable Model to Detect AI-Generated Code. *Information* **2024**, *15*, 819. [CrossRef]
10. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Professional: Reading, MA, USA, 2018.
11. Murphy-Hill, E.; Parnin, C.; Black, A.P. How We Refactor, and How We Know It. *IEEE Trans. Softw. Eng.* **2012**, *38*, 5–18. [CrossRef]
12. Midolo, A.; Tramontana, E. Refactoring Java Loops to Streams Automatically. In Proceedings of the CSSE '21: International Conference on Computer Science and Software Engineering, Singapore, 22–24 October 2021; pp. 135–139. [CrossRef]
13. Zhang, Z.; Xing, Z.; Xia, X.; Xu, X.; Zhu, L. Making Python code idiomatic by automatic refactoring non-idiomatic Python code with pythonic idioms. In Proceedings of the ESEC/FSE 2022: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 696–708. [CrossRef]
14. Zhang, Z.; Xing, Z.; Zhao, D.; Xu, X.; Zhu, L.; Lu, Q. Automated Refactoring of Non-Idiomatic Python Code with Pythonic Idioms. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2827–2848. [CrossRef]
15. Franklin, L.; Gyori, A.; Lahoda, J.; Dig, D. LambdaFicator: From imperative to functional programming through automated refactoring. In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 1287–1290. [CrossRef]
16. Purnima, N.; Salomi Nelaballi, V.S.P.; Kim, D.K. Deep Learning-Based Code Refactoring: A Review of Current Knowledge. *J. Comput. Inf. Syst.* **2024**, *64*, 314–328. [CrossRef]
17. Khojah, R.; Mohamad, M.; Leitner, P.; de Oliveira Neto, F.G. Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice. *Proc. ACM Softw. Eng.* **2024**, *1*, 1819–1840. [CrossRef]
18. White, J.; Hays, S.; Fu, Q.; Spencer-Smith, J.; Schmidt, D.C. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. In *Generative AI for Effective Software Development*; Springer: Cham, Switzerland, 2024; pp. 71–108. [CrossRef]
19. Chavan, O.S.; Hinge, D.D.; Deo, S.S.; Wang, Y.O.; Mkaouer, M.W. Analyzing Developer-ChatGPT Conversations for Software Refactoring: An Exploratory Study. In Proceedings of the MSR '24: 21st International Conference on Mining Software Repositories, Lisbon, Portugal, 15–16 April 2024; pp. 207–211. [CrossRef]
20. Guo, Q.; Cao, J.; Xie, X.; Liu, S.; Li, X.; Chen, B.; Peng, X. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In Proceedings of the ICSE '24: 46th IEEE/ACM International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024. [CrossRef]
21. Siddiq, M.L.; Roney, L.; Zhang, J.; Santos, J.C.D.S. Quality Assessment of ChatGPT Generated Code and their Use by Developers. In Proceedings of the MSR '24: 21st International Conference on Mining Software Repositories, Lisbon, Portugal, 15–16 April 2024; pp. 152–156. [CrossRef]
22. Tufano, R.; Mastropaolo, A.; Pepe, F.; Dabic, O.; Di Penta, M.; Bavota, G. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In Proceedings of the MSR '24: 21st International Conference on Mining Software Repositories, Lisbon, Portugal, 15–16 April 2024; pp. 571–583. [CrossRef]
23. Tang, Y.; Liu, Z.; Zhou, Z.; Luo, X. ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Trans. Softw. Eng.* **2024**, *50*, 1340–1359. [CrossRef]

24. Gyori, A.; Franklin, L.; Dig, D.; Lahoda, J. Crossing the gap from imperative to functional programming through refactoring. In Proceedings of the ESEC/FSE 2013: 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 543–553. [CrossRef]
25. Midolo, A.; Tramontana, E. Replication Package. Available online: https://github.com/AleMidolo/Refactoring-Loops-in-the-Era-of-LLMs-A-Comprehensive-Study (accessed on 28 August 2025).
26. Kapus, T.; Ish-Shalom, O.; Itzhaky, S.; Rinetzky, N.; Cadar, C. Computing summaries of string loops in C for better testing and refactoring. In Proceedings of the PLDI 2019: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, Phoenix, AZ, USA, 22–26 June 2019; pp. 874–888. [CrossRef]
27. Khatchadourian, R.; Tang, Y.; Bagherzadeh, M. Safe automated refactoring for intelligent parallelization of Java 8 streams. *Sci. Comput. Programm.* **2020**, *195*, 102476. [CrossRef]
28. Midolo, A.; Tramontana, E. An API for Analysing and Classifying Data Dependence in View of Parallelism. In Proceedings of the ICCCM '22: 10th International Conference on Computer and Communications Management, Okayama, Japan, 29–31 July 2022; pp. 61–67. [CrossRef]
29. Stein, B.; Clapp, L.; Sridharan, M.; Chang, B.Y.E. Safe stream-based programming with refinement types. In Proceedings of the ASE '18: 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 565–576. [CrossRef]
30. The PEP Editors. Python Enhancement Proposals. 2000. Available online: https://peps.python.org (accessed on 28 August 2025).
31. Midolo, A.; Penta, M.D. Automated Refactoring of Non-Idiomatic Python Code: A Differentiated Replication with LLMs. In Proceedings of the 2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC), Ottawa, ON, Canada, 27–28 April 2025; pp. 1–11. [CrossRef]
32. Liu, B.; Jiang, Y.; Zhang, Y.; Niu, N.; Li, G.; Liu, H. Exploring the potential of general purpose LLMs in automated software refactoring: An empirical study. *Autom. Softw. Eng.* **2025**, *32*, 26. [CrossRef]
33. Cordeiro, J.; Noei, S.; Zou, Y. An Empirical Study on the Code Refactoring Capability of Large Language Models. *arXiv* **2024**, arXiv:2411.02320. [CrossRef]
34. Pomian, D.; Bellur, A.; Dilhara, M.; Kurbatova, Z.; Bogomolov, E.; Bryksin, T.; Dig, D. Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method. In Proceedings of the 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), Flagstaff, AZ, USA, 6–11 October 2024; pp. 275–287. [CrossRef]
35. AlOmar, E.A.; Venkatakrishnan, A.; Mkaouer, M.W.; Newman, C.; Ouni, A. How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In Proceedings of the MSR '24: 21st International Conference on Mining Software Repositories, Lisbon Portugal, 15–16 April 2024; pp. 202–206. [CrossRef]
36. Fraser, G.; Arcuri, A. EvoSuite: Automatic test suite generation for object-oriented software. In Proceedings of the ESEC/FSE '11: 19th ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011; pp. 416–419. [CrossRef]
37. Dabic, O.; Aghajani, E.; Bavota, G. Sampling Projects in GitHub for MSR Studies. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 560–564.
38. maven. Mavenrepository. Available online: https://mvnrepository.com/popular (accessed on 2 September 2024).
39. Smith, N.; Van Bruggen, D.; Tomassetti, F. *Javaparser: Visited*; Leanpub: Victoria, BC, Canada, 2017; Volume 10.
40. Li, J.; Li, G.; Li, Y.; Jin, Z. Structured Chain-of-Thought Prompting for Code Generation. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–23. [CrossRef]
41. Li, J.; Zhao, Y.; Li, Y.; Li, G.; Jin, Z. AceCoder: An Effective Prompting Technique Specialized in Code Generation. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–26. [CrossRef]
42. Liu, C.; Bao, X.; Zhang, H.; Zhang, N.; Hu, H.; Zhang, X.; Yan, M. Improving ChatGPT Prompt for Code Generation. *arXiv* **2023**, arXiv:2305.08360. [CrossRef]
43. Raven, O. Magpie. Available online: https://github.com/openraven/magpie (accessed on 3 September 2024).
44. Vertispan. j2clmavenplugin. Available online: https://github.com/Vertispan/j2clmavenplugin (accessed on 3 September 2024).
45. IridiumIdentity. Iridium. Available online: https://github.com/IridiumIdentity/iridium (accessed on 3 September 2024).
46. OpenRefine. Openrefine. Available online: https://github.com/OpenRefine/OpenRefine (accessed on 3 September 2024).
47. Rapiddweller. Rapiddweller-Benerator-ce. Available online: https://github.com/rapiddweller/rapiddweller-benerator-ce (accessed on 3 September 2024).

48. Liu, F.; Liu, Y.; Shi, L.; Huang, H.; Wang, R.; Yang, Z.; Zhang, L. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv* **2024**, arXiv:2404.00971. [CrossRef]
49. Alkaissi, H.; McFarlane, S.I. Artificial hallucinations in ChatGPT: Implications in scientific writing. *Cureus* **2023**, *15*, e35179. [CrossRef] [PubMed]