Code readability assessment and improvement: human and automated perspectives

Carlos Eduardo de Carvalho Dantas
FACOM, Universidade Federal de Uberlândia
Uberlândia, Brazil
carloseduardodantas@iftm.edu.br

Marcelo de Almeida Maia
FACOM, Universidade Federal de Uberlândia
Uberlândia, Brazil
marcelo.maia@ufu.br

Abstract

Code readability is a fundamental characteristic of software quality. However, challenges remain in both its assessment and the automation of readability improvements. This article investigates code readability from complementary human and automated perspectives. First, it evaluates the readability of code generated by Large Language Models (LLMs) compared to solutions written by developers. Second, it characterizes how developers improve readability in GitHub repositories through the construction of a taxonomy of readability improvements. Finally, it proposes and evaluates a hybrid approach that combines static analysis tools and LLMs to automatically improve code readability. A unified taxonomy comprising 56 types of readability improvements observed in Java and Python projects was developed. The results also show that the code generated by LLMs frequently exhibits fewer readability issues than human-written examples. Additionally, the combination of static analysis recommendations and LLM-based refactorings proved effective in reducing readability issues. Together, these findings advance our understanding of code readability and contribute to the development of tools to improve it.

Keywords

code readability, large language models, pull request, code review, automatic static analysis tools

1 Introduction

Code readability is a fundamental characteristic of software quality because developers spend a substantial portion of their time reading and understanding source code rather than writing it [22, 37]. Readable code contributes to easier maintenance, faster comprehension, reduced error rates, and greater consistency during software evolution [1, 12, 14]. Since software systems continuously evolve through incremental modifications performed by different developers [4, 5, 18], improving readability remains an important concern for both researchers and practitioners.

To address this challenge, several approaches have been proposed. Readability models aim to assess whether the source code is readable [6, 10, 29, 31–33], while automatic static analysis tools (ASATs), such as linters (e.g., SonarLint), identify coding issues that can negatively affect readability [3, 7, 20, 35]. However, previous studies have shown limitations in both approaches. Many warnings reported by automated tools are not addressed in real-world projects [21], some readability improvements performed by developers are not detected [25], and existing readability models do not always reflect the types of improvements observed in real-world software projects [11, 26, 28, 31].

Despite extensive research on readability assessment, there is limited evidence on the relationship between how readability is assessed, how developers improve readability in real-world projects, and how these improvements can be automated. Furthermore, most existing studies focus on assessing readability rather than understanding how developers actually improve readability in real-world software projects [6, 11, 26, 29, 31, 33].

More recently, Large Language Models (LLMs) such as ChatGPT, Gemini, and DeepSeek have emerged as promising tools for software development. Beyond code generation, these models can help developers in understand, review, and refactor source code [13, 24]. Their widespread adoption is evidenced by their increasing presence in GitHub commits, pull requests (PRs), and software development workflows [34, 38, 39]. Despite this growing relevance, important questions remain regarding the readability of LLM-generated code, the extent to which LLMs can improve readability automatically, and how their recommendations compare to the improvements actually performed by developers in real-world projects.

This study investigates code readability from human and automated perspectives. Specifically, (i) evaluates the readability of code generated by LLMs, (ii) characterizes how developers improve readability in real-world GitHub repositories through the construction of a taxonomy of readability improvements, and (iii) proposes and evaluates a hybrid approach that combines static analysis tools and LLMs to automatically improve source code readability.

The main contributions of this work are:

(1) A taxonomy of code readability improvements performed by developers in Java and Python repositories;
(2) A taxonomy of readability-related rules reported by SonarLint static analysis tool;
(3) An empirical comparison between the readability of code generated by LLMs and human-written solutions from Stack Overflow;
(4) A hybrid approach combining ASAT recommendations and LLM-based refactoring to automatically improve code readability;
(5) A replication package containing all the datasets and scripts used in the study [9].

2 Study Design

This article investigates code readability from complementary human and automated perspectives. The overall objective is to understand how code readability can be assessed and improved in modern software development environments, considering both developer practices and LLMs.

To address this objective, the article is organized around the following research questions:

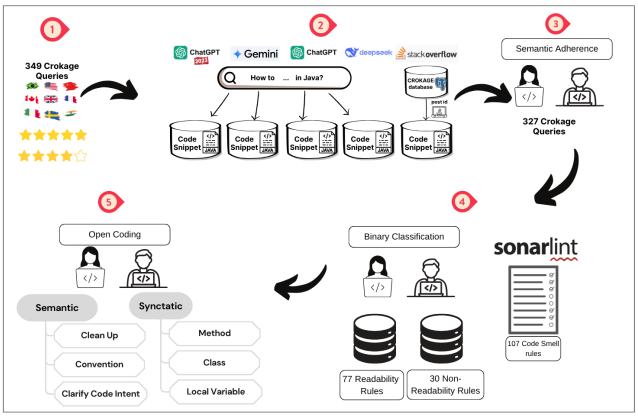

Figure 1: Overview of the proposed methodology to answer RQ1

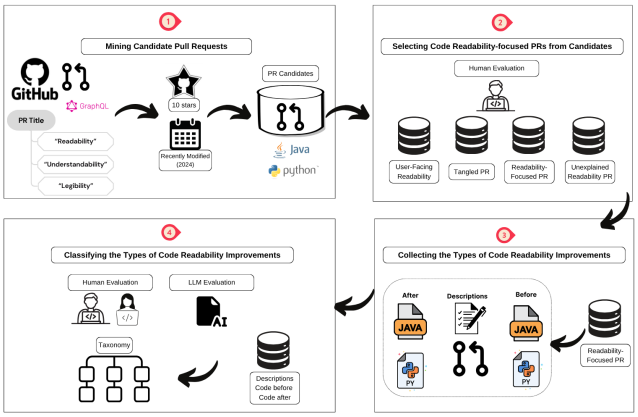

Figure 2: Overview of the proposed methodology to answer RQ2

- **RQ1: How does the readability of the code snippets recommended by LLMs compare to those authored by human developers?**
  This research question aims to investigate whether LLMs are capable of generating code snippets with higher readability compared to those authored by human developers on platforms like Stack Overflow.
- **RQ2: What types of code readability improvements do developers describe and implement in Python and Java Pull Requests (PRs)?**
  This research question aims to propose a taxonomy, identifying and categorizing the different types of code readability improvements made in Python and Java code. The objective is to understand the key readability factors that Python and Java developers address when they submit improvements to their PRs.
- **RQ3: Can LLMs improve code readability based on SonarLint warning recommendations?**
  This question investigates whether LLMs can remove SonarLint-identified code readability warnings without introducing new ones.

The following subsections describe the process used to answer the three research questions.

2.1 Assessing the Readability of LLM-Generated Code

Figure 1 illustrates the methodology proposed to address RQ1. Basically, it is divided into five steps:

2.1.1 Selecting Developer Input Queries. To assess the readability of the LLM-generated code, 349 programming queries were selected from the CROKAGE dataset [8]. These queries were associated with highly rated Stack Overflow recommendations and represent programming tasks considered useful by developers. For each query, equivalent prompts were submitted to ChatGPT, Gemini, and DeepSeek, enabling a comparison between LLM-generated and human-written code snippets.

2.1.2 Code Snippet Extraction from Stack Overflow and LLM Models. For each of the 349 selected queries, code snippets were collected from two sources: Stack Overflow recommendations retrieved by CROKAGE and solutions generated by ChatGPT, Gemini, and DeepSeek. All snippets addressed the same Java programming tasks, enabling a direct comparison between human-written and LLM-generated code.

2.1.3 Assessing Semantic Adherence of Code Snippets with Input Queries. Before assessing readability, the semantic adherence of all code snippets was manually evaluated. Two evaluators independently analyzed whether each Stack Overflow recommendation and LLM-generated solution adequately addressed the corresponding programming query. Only code snippets considered semantically aligned with the query were retained for the study. Following this process, 327 of the 349 queries were considered valid, while 22 were discarded due to insufficient semantic alignment.

2.1.4 Identifying Code Readability Warnings. The 1,635 validated code snippets were analyzed using SonarLint to identify potential readability issues. A custom client was developed to automatically process the snippets and extract the reported code smells. The detected SonarLint rules were then independently classified by two evaluators as readability-related or non-readability-related, resulting in a final set of 77 readability rules and 1,458 readability warnings used in the study.

2.1.5 Classifying Types of Code Readability Rules in SonarLint. The 77 readability-related SonarLint rules were manually analyzed through an open coding process, resulting in a taxonomy of 21 subcategories grouped into five categories. In addition, the syntactic scope of each rule was identified to characterize the code elements most frequently associated with readability warnings.

2.2 Characterizing Readability Improvements in Real-World Projects

Figure 2 illustrates the methodology proposed to address RQ2. It is divided into four steps:

2.2.1 Mining Candidate Pull Requests (PRs). To identify readability-focused PRs, GitHub repositories written in Java and Python were searched using the keywords *readability*, *understandability*, and *legibility* in PR titles. Candidate PRs were filtered to exclude repositories with limited activity, forked projects, non-source-code modifications, and PRs without evidence of peer review. After applying these criteria and manually removing false positives, the final dataset comprised 577 Java and 619 Python candidate PRs for further analysis.

2.2.2 Selecting Readability-focused PRs from Candidates. To reduce subjectivity during the analysis, candidate PRs were manually classified into four categories: readability-focused, user-facing readability, tangled changes, and unexplained readability improvements. Only readability-focused PRs were included, in which developers explicitly described readability-related modifications without combining them with other maintenance activities. This process resulted in a final dataset of 264 Python and 276 Java PRs dedicated to improving source code readability.

2.2.3 Collecting the Types of Code Readability Improvements. For each readability-focused pull request, the readability improvements explicitly described by developers were extracted and validated against the corresponding source code changes. Only confirmed improvements were retained, resulting in 893 readability improvement instances (437 from Python and 456 from Java). These instances, consisting of developer descriptions and the corresponding code changes, formed the dataset used to construct the readability improvement taxonomy.

2.2.4 Classifying the Types of Code Readability Improvements. The 893 readability improvement instances were independently analyzed by two experienced annotators through an open coding process. Initially, labels were assigned to each improvement and iteratively refined as new patterns emerged. To support this process, GPT-4.1 was used as a complementary annotator, helping identify potential disagreements and ambiguous cases that required further review by human evaluators. After resolving conflicts and consolidating semantically similar improvements, the final taxonomy comprised 56 types of readability improvements organized into six categories, including language-independent and language-specific practices. In this process, 19 instances were discarded, resulting in 874 instances.

2.3 LLM-Guided Readability Refactoring

To investigate whether LLMs can automatically improve code readability, the 611 code snippets containing readability warnings identified in RQ1 were used as input. These snippets originated from both Stack Overflow recommendations and LLM-generated solutions and contained at least one readability-related warning reported by SonarLint. For each affected snippet, the corresponding SonarLint recommendations were extracted and incorporated into a prompt requesting a readability-oriented refactoring.

The code snippets and their associated warnings were submitted to GPT-4.1, Gemini 2.5 Pro, and DeepSeek Coder through their respective APIs, resulting in 1,833 refactored code snippets. In this scenario, API-accessible models were selected because automated large-scale refactoring required programmatic access. The prompt instructed the models to revise the code according to the readability recommendations while preserving the original functionality. All requests were executed with MAX_TOKENS = 8,192, that is, each request was limited to this number of tokens and TEMPERATURE = 0.0, as previous work has shown that lower temperature settings tend to produce better and more stable results for code refinement tasks [13]. The generated snippets were then reanalyzed using SonarLint to determine which warnings had been removed and whether new warnings had been introduced.

Prompt Template

System: "You are a software developer specialized in the Java programming language, with expertise in improving the readability of the code based on a list of recommendations."
User: "Given the following Java code snippet: `[CODE_SNIPPET]`, the following improvements are recommended: `[SONARLINT_WARNINGS]`. Please provide a revised version of the code snippet that applies the recommended improvements. Only the revised code snippet, without additional text."

Finally, the effectiveness of the three models was evaluated from two perspectives: (i) the number of readability warnings successfully removed, and (ii) the number of code snippets completely corrected without introducing new issues. Statistical comparisons were conducted using Friedman and Nemenyi tests for warning counts, and Cochran's Q followed by McNemar tests for binary correction outcomes.

Figure 3 illustrates this process. In the query with ID = 26, the code snippets generated by ChatGPT, DeepSeek, and Stack Overflow did not raise any readability warnings. However, Gemini raised five warnings and ChatGPT 3.5 raised one. Then these code snippets were submitted to the three LLMs in an attempt to address the detected warnings. The warning raised by ChatGPT 3.5 was eliminated by all three LLMs. In contrast, regarding the warnings raised by Gemini, only the version generated by Gemini 2.5 Pro itself managed to remove them. DeepSeek Coder failed to resolve the java:S135 warning, while GPT-4.1 did not fix the java:S135 or java:S3776 warnings and even introduced a new one: java:S3626.

3 Results

This section describes the results according to each research question.

3.1 RQ1: How does the readability of the code snippets recommended by LLMs compare to those authored by human developers?

Table 1 shows that LLM code snippets contain substantially fewer readability warnings than human-written solutions from Stack Overflow. While 66% of the Stack Overflow snippets were affected by at least one readability warning, the proportion ranged from 24% to 34% among the LLMs evaluated. Similarly, Stack Overflow accumulated 623 readability warnings, compared to 225 for ChatGPT 2023, 151 for ChatGPT 2025, 261 for DeepSeek, and 198 for Gemini.

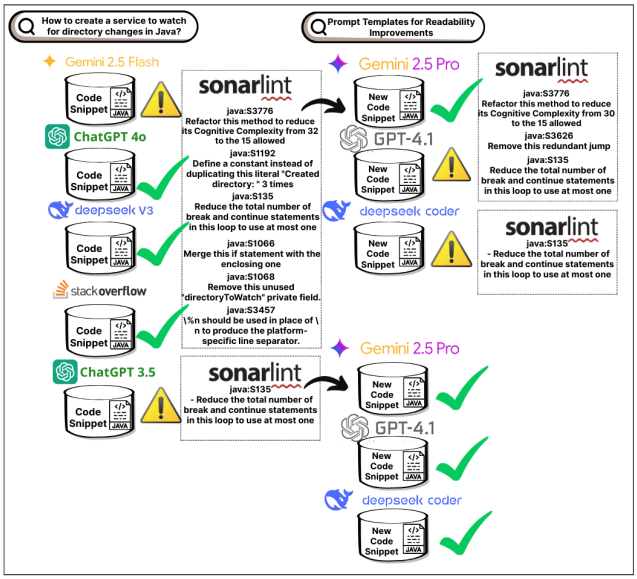

Figure 3: Improving Code Readability Example on Query ID = 26

The Friedman test was performed using a confidence level of 95% (p-value < 0.05), and the post hoc Nemenyi tests confirmed that Stack Overflow contains significantly more readability warnings compared to all LLMs, while ChatGPT 2025 produced significantly fewer warnings than DeepSeek.

Table 1: Readability warnings detected by SonarLint (n = 327 snippets per baseline).

Baseline	Affected	%	Warnings
Stack Overflow	216	66%	623
ChatGPT 2023	97	29%	225
ChatGPT 2025	80	24%	151
DeepSeek	114	34%	261
Gemini	104	31%	198

Additional analyses were performed using lines of code (LOC), cognitive complexity, and the readability model proposed by Scalabrino et al. In general, Gemini generated significantly longer code snippets, included more comments, and exhibited higher cognitive complexity than the other baselines. Regarding readability classification, most of the snippets from all baselines were considered readable, with rates ranging from 68% to 84%. However, the Chi-squared test revealed that Stack Overflow contained more unreadable snippets than expected, while ChatGPT 2025 produced fewer unreadable snippets than expected. Correlation analyses revealed only weak associations between SonarLint readability warnings and evaluated metrics, suggesting that readability warnings capture aspects of code quality that are not fully reflected by traditional readability and complexity measures.

The readability warnings were grouped into five categories. As shown in Table 2, *Clean Up* was the most frequent category in all baselines, accounting for the majority of the detected warnings. Stack Overflow accumulated substantially more warnings than the evaluated LLMs, particularly in the *Clean Up* and *Convention* categories. Among the LLMs, ChatGPT 2025 produced the lowest number of readability warnings, while DeepSeek exhibited the highest number of warnings after Stack Overflow.

Table 2: Readability warnings by category and baseline.

Category	SO	GPT23	GPT25	DS	Gem
Clarify Code Intent	90	24	26	31	54
Prefer Concise Code	91	35	12	17	26
Clean Up	303	155	98	209	97
Convention	130	7	11	3	14
Improve Modularity	9	4	4	1	7
Total	623	225	151	261	198

3.2 RQ2: What types of code readability improvements do developers describe and implement in Python and Java Pull Requests (PRs)?

The open coding process applied to the 426 Python and 448 Java instances (total of 874 instances) under 260 and 274 PRs respectively, identified 38 different types of changes for Python and 44 for Java, grouped into six categories: Clarify Code Intent, Prefer Concise Code, Clean Up, Improve Formatting, Improve Code Modularity and Improve Documentation.

Table 3 summarizes the categorization of Java and Python PRs. Both languages show a strong focus on clarifying code intent. However, while Java PRs tend to emphasize the elimination of redundant or unnecessary code, Python PRs more often focus on improving documentation and code formatting according to style guidelines.

Table 3: Distribution of readability-focused PRs by category.

Category	Python	Java
Clarify Code Intent	125 (48%)	125 (46%)
Prefer Concise Code	80 (30%)	96 (35%)
Clean Up	23 (9%)	43 (16%)
Improve Code Modularity	38 (14%)	51 (19%)
Improve Documentation	65 (25%)	38 (14%)
Improve Code Formatting	41 (16%)	21 (8%)
Total PRs	260	274

3.2.1 Clarify Code Intent. Focuses on making the code more intuitive and self-explanatory, thus reducing the subjectivity involved in how other developers read and understand it. Unlike the Prefer Concise Code or Clean Up category, the improvements here do not necessarily involve removing tokens, but rather clarifying the intention of the code. Table 4 presents the types of readability improvements in the category Clarify Code Intent for Python and Java.

3.2.2 Prefer Concise Code. Generally, developers describe verbosity as the presence of code that includes tokens that are not necessary to solve a task. Reducing code verbosity involves eliminating these superfluous elements, resulting in more concise and

Table 4: Top readability improvements in the Clarify Code Intent category.

Improvement	Python	Java	Total
Improve Naming	50	42	92
Include Modifiers and Annotations	4	25	29
Extract Literals to Constants	7	16	23
Extract Variable	14	6	20
Clarify Test Intent	6	10	16
Specialized API Usage	9	7	16
Prefer Positive Conditions	11	4	15
Type Hints / Named Parameters.	11	0	11

readable implementations. In general, verbosity occurs when developers write additional code that is not essential for implementing a given feature. Table 5 presents the most frequent readability improvements in the Prefer Concise Code category.

Table 5: Top readability improvements in the Prefer Concise Code category.

Improvement	Python	Java	Total
Remove Duplicate Code	18	10	28
Improve String Formatting	21	3	24
Prefer Built-in Methods	5	19	24
Reduce Conditional Nesting	11	9	20
Simplify Object Construction	8	12	20
Unwrap Unnecessary Else	9	7	16
Prefer Idiomatic Iteration	6	10	16
Replace Anonymous Classes	0	12	12
Collapse Boolean Return	4	6	10

3.2.3 Clean Up. While Prefer Concise Code removes unnecessary additional code to implement a feature, this category removes unused code, that is, a custom method never called or a field never used. Table 6 illustrates the results of clean up types of readability improvements for Python and Java.

Table 6: Readability improvements in the Clean Up category.

Improvement	Python	Java	Total
Remove Unused Chunk of Code	15	10	25
Remove Commented Code	0	14	14
Remove Unnecessary Throws	0	6	6
Remove Unused Imports	3	5	8
Remove Useless Parameters	0	4	4
Remove Unnecessary Modifier	0	4	4
Remove Unused Field/Variable	0	3	3
Others	5	1	6

3.2.4 Improve Code Modularity. Modularity refers to the separation of the complexity of a program into smaller and more manageable chunks of code [2]. It differs from other categories in that it does not involve polishing a code snippet as seen in categories such as Prefer Concise Code, or removing unused code as described in the Clean Up category. Although improving modularity may also contribute to clarifying the intention of the code, it also has a motivation to organize the codebase into coherent classes and methods. Table 7 illustrates the results of types of readability improvements for Python and Java.

Table 7: Readability improvements in the Improve Code Modularity category.

Improvement	Python	Java	Total
Extract Method from Large Function	19	32	51
Move Logic to Another File	10	15	25
Extract Lambda to a Named Function	5	0	5
Extract Logic to Dataclass	4	0	4
Use Builder Pattern	0	4	4
Others	0	2	2

3.2.5 Improve Documentation. The aim is to improve code readability using comments [36] and documentation such as Python docstrings. Table 8 illustrates the results of types of readability improvements for Python and Java.

Table 8: Readability improvements in the Improve Documentation category.

Improvement	Python	Java	Total
Clarify Comments/Docstrings	42	22	64
Add New Comments/Documentation	28	29	57
Others	1	0	1

3.2.6 Improve Code Formatting. Previous readability models have already addressed code formatting such as line length and indentation [6, 10, 29]. This improvement aims to visually group related code blocks and make them easier to distinguish from each other. Table 9 illustrates the results of types of readability improvements for Python and Java.

Table 9: Readability improvements in the Improve Formatting category.

Improvement	Python	Java	Total
Use PEP Guidelines	14	0	14
Format Spacing (Python Black)	13	0	13
Reorder Elements	7	9	16
Limit Line Length	8	6	14
Standardize Formatting	0	5	5
Others	2	1	3

3.3 RQ3: Can LLMs improve code readability based on SonarLint warning recommendations?

Table 10 presents the number of readability warnings before and after correction by each LLM. The 611 code snippets affected by

readability warnings, as identified in RQ1, contained a total of 1,109 warnings. GPT-4.1 reduced this number to 281, achieving a reduction of 74.7%, while DeepSeek Coder achieved a reduction of 82.6%, and Gemini 2.5 Pro a reduction of 81.2%. In particular, both DeepSeek Coder and Gemini 2.5 Pro eliminated a substantially larger proportion of warnings compared to GPT-4.1. This result is particularly interesting given that the free version of ChatGPT produced code snippets with the fewest readability warnings in RQ1, whereas GPT-4.1 accessed via the API showed the weakest performance when refactoring those snippets based on SonarLint recommendations, especially in the *Clean Up* category.

To verify whether the remaining warnings differed significantly between models, the Friedman statistical test was applied. The test returned $p = 6.51 \times 10^{-7}$, rejecting the null hypothesis and confirming that there are differences. A subsequent Nemenyi post hoc test indicated that the difference between DeepSeek Coder and GPT-4.1 was nearly significant ($p = 0.06$).

Another relevant aspect is the reduction rate per category, which shows that LLMs eliminated the vast majority of warnings related mainly to *Clarify Code Intent* and *Prefer Concise Code*. Gemini 2.5 Pro proved to be more effective than the others in reducing warnings related to conventions and modularity, whereas DeepSeek Coder performed better in the *Clean Up* category.

Table 11 presents the number of code snippets affected by readability warnings before and after correction by each LLM. In GPT-4.1, 179 code snippets are still affected by readability warnings, which corresponds to a reduction of 70.7%. DeepSeek Coder achieved a reduction of 79.4%, while Gemini 2.5 Pro achieved a reduction of 77.7%. Similarly to the overall number of readability warnings, both DeepSeek Coder and Gemini 2.5 Pro eliminated a considerably larger proportion of affected snippets compared to GPT-4.1.

Statistical analyzes revealed significant differences between the LLMs evaluated in terms of their ability to automatically improve code readability. The Cochran's Q test indicated that the proportion of fully corrected code snippets differed significantly between models ($p < 0.01$). The pairwise McNemar tests showed that both DeepSeek Coder and Gemini 2.5 Pro corrected significantly more code snippets than GPT-4.1, whereas no significant difference was observed between DeepSeek Coder and Gemini 2.5 Pro. These results suggest that code-oriented LLMs can be highly effective in resolving readability warnings, although their effectiveness varies substantially between models.

3.3.1 Introduction of New Readability Warnings. Although the previous subsections provided information on the reduction of warnings and the resolution of affected code snippets, they did not address the perspective of newly introduced warnings that were not present in the original code. For example, among the 611 snippets affected by readability warnings, 370 contained issues related to the *Clean Up* category. After applying LLMs to remove these warnings, 148 snippets still exhibited *Clean Up* issues. However, a portion of these 148 cases corresponded to snippets that were not among the original 370, indicating that new warnings were introduced during the correction process.

Table 12 illustrates the number of newly introduced readability warnings and affected code snippets for each LLM. GPT-4.1 introduced 105 new warnings in 89 snippets, DeepSeek Coder introduced 67 across 58 snippets, and Gemini 2.5 Pro introduced 75 across 71 snippets.

Combining the information from Tables 10, 11, and 12, it can be observed that GPT-4.1 ended with 281 readability warnings in total, of which 105 were newly introduced, while the remaining 176 correspond to original warnings that were not corrected. A similar reasoning applies to the number of code snippets: 179 snippets still contained readability warnings, 89 of which had newly introduced warnings, whereas the other 90 correspond to cases where existing warnings were not fully resolved and no new warnings were introduced.

3.4 Discussion of the Results

The results reveal an interesting contrast between human and automated perspectives on code readability. Although modern LLMs generated code with substantially fewer readability warnings than human-written solutions from Stack Overflow, the taxonomy extracted from GitHub PRs showed that developers frequently perform readability improvements that go beyond the scope of traditional static analysis tools. In particular, developers often focus on clarifying code intent through meaningful naming, semantic refinements, documentation, and language-specific conventions. These findings suggest that readability cannot be fully characterized by syntactic metrics or linter warnings alone.

Another important observation is that LLMs appear to reproduce coding conventions that are broadly adopted by the software development community. This behavior explains their strong performance in readability-related rules involving naming conventions (e.g., Java camelcase), code patterns (e.g., Java indentation and Javadoc comments), and concise implementations. However, the results also indicate that LLMs do not explicitly optimize for readability. Instead, they primarily aim to generate a correct solution to the developer's query, often reproducing patterns observed during training. As a consequence, some readability improvements require explicit guidance through prompts or recommendations derived from static analysis tools (e.g., explicitly requesting a Java lambda instead of a traditional `for-each` loop).

The three studies suggest that static analysis tools and LLMs provide complementary strengths. Linters are effective in identifying objective readability issues, while LLMs can use contextual information to automatically refactor code and resolve many of these warnings. At the same time, the taxonomy of real-world improvements highlights several readability practices that remain difficult to capture through automated rules alone, particularly those involving semantics and developer intent. These findings support a hybrid vision in which readability-aware development tools combine static analysis, developer knowledge, and LLM-based reasoning to assess and improve source code readability.

3.5 Implications for Researchers and Developers

For researchers, the results show that combining LLMs with linters to improve code readability can lead to effective refactorings, as long as the process is properly guided. Using a lower temperature parameter tends to make the LLM more deterministic and predictable, but also often literal. As shown in the analysis, when

Table 10: Total number of readability warnings before refactoring and after correction by each LLM, along with the percentage reduction relative to the baseline. ($n = 1,109$ readability warnings)

Category	Before	GPT-4.1		DeepSeek Coder		Gemini 2.5 Pro	
	Count	Count	Reduction	Count	Reduction	Count	Reduction
Clean Up	635	220	65.4%	153	75.9%	193	69.6%
Clarify Code Intent	179	15	91.6%	11	93.9%	6	96.6%
Prefer Concise Code	158	18	88.6%	7	95.6%	3	98.1%
Convention	116	22	81.0%	16	86.2%	5	95.7%
Improve Code Modularity	21	6	71.4%	6	71.4%	2	90.5%
Total	1109	281	74.7%	193	82.6%	209	81.2%

Table 11: Total number of affected code snippets before refactoring and after correction by each LLM, along with the percentage reduction relative to the baseline. ($n = 611$ snippets)

Category	Before	GPT-4.1		DeepSeek Coder		Gemini 2.5 Pro	
	Count	Count	Reduction	Count	Reduction	Count	Reduction
Clean Up	370	148	60.0%	99	73.2%	125	66.2%
Clarify Code Intent	169	14	91.7%	11	93.5%	6	96.4%
Prefer Concise Code	156	16	89.7%	7	95.5%	3	98.1%
Convention	106	17	84.0%	15	85.8%	5	95.3%
Improve Code Modularity	19	3	84.2%	4	78.9%	1	94.7%
Total	611	179	70.7%	126	79.4%	136	77.7%

Table 12: Introduced readability warnings by category and LLM (counts of warnings and affected snippets). Totals reflect unique snippets overall and may not equal the sum across categories.

Category	GPT-4.1		DeepSeek Coder		Gemini 2.5 Pro	
	Warn.	**Snip.**	**Warn.**	**Snip.**	**Warn.**	**Snip.**
Clean Up	79	72	50	46	69	66
Clarify Code Intent	5	5	1	1	1	1
Prefer Concise Code	12	12	6	6	2	2
Convention	8	8	8	8	2	2
Improve Code Modularity	1	1	2	2	1	1
Total	**105**	**89**	**67**	**58**	**75**	**71**

instructed to remove a specific element, the model simply removes it, often without considering that the change might raise another readability warning. This highlights the need for better prompt design to avoid long refinement cycles between LLMs and developers. These findings open opportunities for developing new IDE plugins that combine understanding the context of the code with static analysis rules to guide the refactoring process more effectively.

For developers and practitioners, this work identified several cases in which developers improve code readability by adopting features introduced through the evolution of Python and Java. Examples include `try-with-resources`, lambdas, and pattern matching. These findings emphasize the importance of embracing modern language features to enhance code readability and take advantage of language advances. Moreover, code snippets extracted from platforms such as question-and-answer sites may exhibit a considerably higher number of readability warnings, especially when authored by individuals with limited experience in Java who may be unfamiliar with some of its key coding conventions.

4 Threats to Validity

The findings should be interpreted considering some limitations. Regarding construct validity, code readability can be perceived differently by developers and cannot be fully captured by a single assessment approach. To mitigate this threat, the article combines complementary perspectives, including readability warnings reported by SonarLint, improvements explicitly performed by developers in GitHub PRs, and evaluations involving LLMs. Nevertheless, some aspects of readability, particularly those involving semantic understanding, naming quality, and developer intent, may not be

fully represented by the adopted instruments. In addition, the results depend on the selected datasets and tools, including SonarLint, CROKAGE, GitHub repositories, and LLM-based annotations.

Regarding internal validity, parts of the data collection and classification process required manual intervention, including the extraction of code snippets, the classification of readability improvements, and the assessment of semantic adherence. To reduce potential biases and errors, the study employed multiple annotators, manual reviews, and consensus-based resolution of disagreements. Another limitation concerns the automated refactorings generated by LLMs. Although readability warnings were reduced, the study did not perform functional validation through automated test suites, limiting the ability to guaranty behavioral preservation in all cases.

Finally, in external validity, the results are based on Java and Python datasets, as well as specific versions of ChatGPT, Gemini, and DeepSeek. Although these languages and models are widely adopted in real-world projects, the findings may not be directly generalized to other programming languages, software ecosystems, or future generations of LLMs. Given the rapid evolution of AI generative models, future studies may observe different results as new models, prompting strategies, and development tools emerge.

5 Related Work

Previous studies have investigated code readability from different perspectives. Readability models, such as the one proposed by Buse and Weimer [6] and several subsequent models [10, 29, 32, 33], aim to automatically classify source code according to its readability. Some of these models have also been incorporated into code search engines to rank code snippets according to their readability [15, 16, 23]. However, subsequent studies have reported that existing readability metrics often fail to capture improvements performed by developers in real-world repositories [11, 26, 31]. Given that prior studies have reported that existing metrics are not effectively able to capture what developers actually consider code readability, this limitation motivated the proposal of a new taxonomy in this work.

From the developer perspective, Pantiuchina et al. [27] showed that readability and understandability are among the most common motivations for refactoring activities. More recently, Oliveira et al. [25] proposed a taxonomy of understandability-related issues extracted from code review comments and observed that many of these issues are not detected by existing linters, suggesting a gap between automated recommendations and developer perceptions. The taxonomy proposed in this work includes examples from the Python programming language and explores differences in languages from distinct paradigms. Furthermore, this work also focuses on merged PR explicitly aimed at improving code readability.

Static analysis tools have also been investigated as a mechanism for identifying code quality and readability issues. Felipe et al. [7] analyzed technical debts in merged PRs from 12 Apache Java projects using the SonarQube tool. The connection between this study and the present work lies in the shared reliance on Sonar-based tools to extract rule violations. However, this work takes a more specific approach, focusing specifically on code smells that impact readability. Romano et al. [30] showed that the use of SonarLint can reduce code smells and improve code understandability, while [19] conducted a large-scale investigation to assess which types of code quality issues can be effectively detected by six ASAT tools, with the precision of the tools varied considerably, ranging from 18% to 86%.

Other studies have explored automated approaches for improving source code quality. Loriot et al. [20] proposed Styler, a neural approach guided by Checkstyle rules to automatically repair style violations. Guo et al. [13] investigated the use of ChatGPT for automated code refinement based on real code review scenarios, showing that model configuration and the quality of review suggestions influence the effectiveness of generated refinements. More recently, Jaoua et al. [17] investigated a hybrid approach that combined linters and LLMs to improve code review comments. These studies demonstrate the potential of combining automated recommendations with neural or generative models. This work investigates the broader relationship between code readability, developer practices, and LLM-based readability improvement.

6 Conclusions and Future Work

This article investigated code readability from complementary human and automated perspectives. In general, the results provide complementary evidence on the different dimensions involved in assessing and improving code readability. The LLMs-generated code contained significantly fewer readability warnings than the human-written code extracted from Stack Overflow, while the LLMs guided by recommendations of static analysis were able to address a substantial proportion of the identified readability issues. From a human perspective, the taxonomy of 56 types of readability improvements extracted from real-world Java and Python projects showed that developers frequently perform changes that go beyond syntactic rules, involving code intent, semantic understanding, and developer judgment.

These findings support a broader hybrid view of readability improvement in which static analysis, developer knowledge, and LLM-based reasoning play complementary roles. Rather than replacing one another, these perspectives address different dimensions of readability: static analysis provides explicit and verifiable rules, developers reveal the readability practices that emerge in real-world software evolution, and LLMs provide contextual reasoning that can help automate readability-oriented transformations. Therefore, effective readability support should move beyond isolated metrics or syntactic rules toward hybrid approaches that combine these complementary capabilities.

Future work includes the development of a readability-oriented prompt engineering framework that integrates linter recommendations, language-specific conventions, and readability patterns extracted from real-world repositories. Additional research directions include evaluating semantic aspects of readability improvements, such as identifier naming and documentation quality, extending the analysis to other programming languages, and investigating the use of LLM-based agents capable of autonomously identifying and improving readability issues in software repositories.

Artifact Availability

The artifacts are publicly available on the GitHub repository https://github.com/carloseduardoxp/thesis

References

[1] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. 2002. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*. 235–241. https://doi.org/10.1109/RAMS.2002.981648

[2] Fabian Beck and Stephan Diehl. 2011. On the Congruence of Modularity and Code Coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 354–364. doi:10.1145/2025113.2025162

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 470–481. https://doi.org/10.1109/SANER.2016.105

[4] Hans Christian Benestad, Bente Anda, and Erik Arisholm. 2009. Understanding software maintenance and evolution by analyzing individual changes: a literature review. *J. Softw. Maint. Evol.* 21, 6 (Nov. 2009), 349–378. https://doi.org/10.1002/smr.412

[5] Keith H. Bennett and Václav T. Rajlich. 2000. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (Limerick, Ireland) *(ICSE '00)*. Association for Computing Machinery, New York, NY, USA, 73–87. doi:10.1145/336512.336534

[6] Raymond P.L. Buse and Westley R. Weimer. 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558. doi:10.1109/TSE.2009.70

[7] Felipe Calixto, Eliane Araújo, and Everton Alves. 2024. How does Technical Debt Evolve within Pull Requests? An Empirical Study with Apache Projects. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 212–223. https://doi.org/10.5753/sbes.2024.3368

[8] Rodrigo Fernandes Gomes da Silva, Chanchal K. Roy, Mohammad Masudur Rahman, Kevin A. Schneider, Klérisson V. R. Paixão, Carlos Eduardo de Carvalho Dantas, and Marcelo de Almeida Maia. 2020. CROKAGE: effective solution recommendation for programming tasks by leveraging crowd knowledge. *Empir. Softw. Eng.* 25, 6 (2020), 4707–4758. https://doi.org/10.1007/s10664-020-09863-2

[9] Carlos Eduardo C. Dantas and Marcelo A. Maia. 2025. Code Readability Assessment and Improvement: Human and Automated Perspectives. https://github.com/carloseduardoxp/thesis.

[10] Jonathan Dorn. 2012. A General Software Readability Model. https://api.semanticscholar.org/CorpusID:14098740

[11] Sarah Fakhoury, Devjeet Roy, Sk. Adnan Hassan, and Venera Arnaoudova. 2019. Improving Source Code Readability: Theory and Practice. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, 2–12. doi:10.1109/ICPC.2019.00014

[12] Prashant Gosavi and Václav T. Rajlich. 2004. Incremental Change in Object-Oriented Programming . *IEEE Software* 21, 04 (July 2004), 62–69. https://doi.org/10.1109/MS.2004.17

[13] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 34, 13 pages. doi:10.1145/3597503.3623306

[14] Nuzhat J. Haneef. 1998. Software documentation and readability: a proposed process improvement. *SIGSOFT Softw. Eng. Notes* 23, 3 (May 1998), 75–77. doi:10.1145/279437.279470

[15] Andre Hora. 2021. Googling for Software Development: What Developers Search For and What They Find. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 317–328. https://doi.org/10.1109/MSR52588.2021.00044

[16] André C. Hora. 2021. APISonar: Mining API usage examples. *Software: Practice and Experience* 51 (2021), 319 – 352. https://doi.org/10.1002/spe.2906

[17] Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. 2025. Combining Large Language Models with Static Analyzers for Code Review Generation . In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, Los Alamitos, CA, USA, 174–186. doi:10.1109/MSR66628.2025.00038

[18] Chris Kemerer. 1995. Software Complexity and Software Maintenance: A Survey of Empirical Research. *Ann. Software Eng.* 1 (12 1995), 1–22. https://doi.org/10.1007/BF02249043

[19] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), 111575. doi:10.1016/j.jss.2022.111575

[20] Benjamin Loriot, Fer Madeiral, and Martin Monperrus. 2022. Styler: learning formatting conventions to repair Checkstyle violations. *Empirical Software Engineering* 27 (08 2022). doi:10.1007/s10664-021-10107-0

[21] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. 2019. Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 209–219. doi:10.1109/ICPC.2019.00040

[22] Roberto Minelli, Andrea Mocci and, and Michele Lanza. 2015. I know what you did last summer: an investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension* (Florence, Italy) *(ICPC '15)*. IEEE Press, 25–35. https://doi.org/10.1109/ICPC.2015.12

[23] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 880–890. https://doi.org/10.1109/ICSE.2015.98

[24] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. doi:10.1145/3597503.3639187

[25] Delano Oliveira, Reydne Santos, Benedito de Oliveira, Martin Monperrus, Fernando Castor, and Fernanda Madeiral. 2025. Understanding Code Understandability Improvements in Code Reviews. *IEEE Transactions on Software Engineering* 51, 1 (2025), 14–37. doi:10.1109/TSE.2024.3453783

[26] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. 2018. Improving Code: The (Mis) Perception of Quality Metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 80–91. https://doi.org/10.1109/ICSME.2018.00017

[27] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why Developers Refactor Source Code: A Mining-based Study. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 29 (Sept. 2020), 30 pages. doi:10.1145/3408302

[28] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. 2020. How does code readability change during software evolution? *Empirical Software Engineering* 25 (11 2020), 1–39. doi:10.1007/s10664-020-09886-9

[29] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (Waikiki, Honolulu, HI, USA) *(MSR '11)*. Association for Computing Machinery, New York, NY, USA, 73–82. doi:10.1145/1985441.1985454

[30] Simone Romano, Fiorella Zampetti, Maria Teresa Baldassarre, Massimiliano Di Penta, and Giuseppe Scanniello. 2022. Do Static Analysis Tools Affect Software Quality when Using Test-driven Development?. In *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (Helsinki, Finland) *(ESEM '22)*. Association for Computing Machinery, New York, NY, USA, 80–91. doi:10.1145/3544902.3546233

[31] Devjeet Roy, Sarah Fakhoury, John Lee, and Venera Arnaoudova. 2020. *A Model to Detect Readability Improvements in Incremental Changes*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/3387904.3389255

[32] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. doi:10.1109/ICPC.2016.7503707

[33] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *Journal of Software: Evolution and Process* 30 (06 2018). doi:10.1002/smr.1958

[34] Julyanara Silva, Carlos Dantas, and Marcelo Maia. 2024. What Developers Ask to ChatGPT in GitHub Pull Requests? an Exploratory Study. In *Anais do XII Workshop de Visualização, Evolução e Manutenção de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 125–136. https://doi.org/10.5753/vem.2024.3913

[35] SONARLINT. 2025. Sonarlint Web Page. https://www.sonarsource.com/products/sonarlint/. Accessed: 2025-02-20.

[36] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 2–13. doi:10.1145/3387904.3389258

[37] Rebecca Tiarks. 2011. What Programmers Really Do - An Observational Study. *Softwaretechnik-Trends* 31 (2011). https://dl.gi.de/server/api/core/bitstreams/e6bef151-fd86-4acd-a2ad-81f66aa4b101/content

[38] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 571–583. doi:10.1145/3643991.3644918

[39] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2024. DevGPT: Studying Developer-ChatGPT Conversations. In *Proceedings of the International Conference on Mining Software Repositories (MSR 2024)*. doi:10.1145/3643991.3648400