How Readable Is LLM-Generated Code Snippets? A Comparison of ChatGPT, DeepSeek, and Gemini

Giovanna Fernandes[1], **Marcelo A. Maia**[2], **Carlos Eduardo C. Dantas**[1]

[1]Instituto Federal do Triângulo Mineiro (IFTM) – Uberlândia, MG – Brazil

[2]Universidade Federal de Uberlândia (UFU) – Uberlândia, MG – Brazil

{giovannafernandes,carloseduardodantas}@iftm.edu.br, marcelo.maia@ufu.br

Abstract. *Developers often search for reusable code snippets on the Web. With the increasing adoption of Large Language Models (LLMs) to support programming tasks and the growing number of available models, this work proposes an evaluation of the readability of 981 code snippets generated by ChatGPT, DeepSeek, and Gemini (327 by each model) by analyzing the warnings detected by static analysis tools such as SonarLint. Additionally, we present a preliminary approach that combines SonarLint recommendations with LLMs to automatically refactor code snippets with the goal of improving code readability. The results show that ChatGPT produces code with fewer readability warnings according to SonarLint. All three LLMs were also able to remove readability warnings in more than 60% of the affected code snippets. However, challenges remain when combining LLMs with static analysis tools, particularly in understanding the context of certain rules and avoiding the removal of relevant code. The insights from this study reveal opportunities for deeper integration between static analysis tools and LLMs.*

1. Introduction

Code snippets, or code examples, are short and reusable fragments that demonstrate how to solve specific programming problems [Keivanloo et al. 2014]. They improve learning [Holmes et al. 2009], promote code reuse, and can accelerate software development. Developers frequently search the web for code examples and accompanying explanations to support their programming tasks [Sadowski et al. 2015], spending up to 20% of their time on this activity [Niu et al. 2017].

Historically, developers often rely on web search engines such as Google to retrieve code snippets [Hora 2021a]. Some previous studies have proposed the development of specialized code search engines that extract code snippets from repositories such as GitHub [Hora 2021b] or question and answer platforms such as Stack Overflow [da Silva et al. 2020].

As large language models (LLMs) continue to revolutionize the way code is written, models such as ChatGPT [ChatGpt 2025] have proven particularly effective for tasks such as code refactoring [Zhang et al. 2024], bug fixing [Sobania et al. 2023], and other programming activities [Tufano et al. 2024, Silva et al. 2024]. Previous research has shown that ChatGPT is capable of generating didactic high-quality code snippets that are often easier to read than those written by humans on Stack Overflow [Dantas et al. 2023].

More recently, new LLMs have been released, such as Google's Gemini [Google 2025] and DeepSeek [DeepSeek 2025]. The latter, in particular, has gained significant media attention due to its development in China and its impact on the financial market, contributing to the drop in the stock prices of major tech companies such as NVIDIA and Oracle [G1 Tecnologia 2025]. Like ChatGPT, both LLMs are also capable of generating source code to address tasks requested by developers.

With the emergence of a new generation of LLMs competing for developers' attention, a key research opportunity is to systematically evaluate the quality of the source code produced by them. In particular, assessing the readability of generated code is fundamental, as it is a key concern for developers during code writing [Piantadosi et al. 2020], who often spend a significant portion of their time reading and understanding code [Minelli et al. 2015].

This study aims to assess the readability of code snippets generated by LLMs such as ChatGPT, Gemini, and DeepSeek and their ability to perform code readability improvements in existing code. These code snippets were obtained from a collection of 327 input queries extracted from a previous work [Dantas et al. 2023]. This study provides the following contributions.

1. A comparative study between the readability of code snippets generated by ChatGPT, Gemini, and DeepSeek.
2. A catalog of 54 SonarLint rules used in this work that improves code readability.
3. A study analyzing to what extent LLMs automatically improve code readability by comparing the correction, persistence, or even the introduction of new readability warnings in code snippets.
4. A replication package that includes scripts, queries, and the corresponding recommended code snippets from DeepSeek, Gemini, and ChatGPT, to support future research [Fernandes et al. 2025].

2. Methodology

This study is driven by the following research questions:

- **RQ_1: How do LLM-generated code snippets differ in readability, as measured by linters such as SonarLint?** This research question investigates the types of code readability warnings identified by linters such as SonarLint [SonarLint 2025] in code snippets generated by different LLMs, comparing the distribution of these readability warnings between models.
- **RQ_2: 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.

Figure 1 illustrates the methodology proposed to answer the research questions. Each step is described in the following subsections.

2.1. Extracting Code Snippets from LLMs

The first step, as shown in Figure 1, includes selecting the input queries from which the code snippets were obtained. These queries were extracted from a previous work

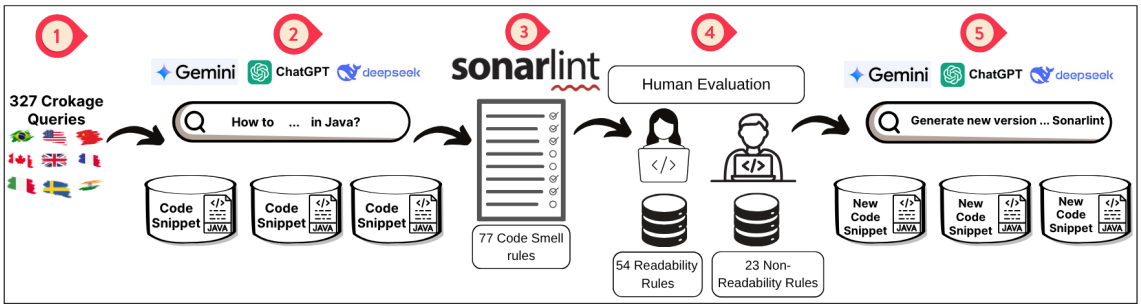

Figure 1. Overview of the proposed methodology

[Dantas et al. 2023] that produced a curated data set of 327 queries from users spanning approximately 80 countries in the CROKAGE code search tool [Crokage 2025].

To obtain the code snippets as illustrated in step 2 of Figure 1, each query was submitted as a developer or computer science student would typically do, that is, by initiating a new chat for each question, appending the token "in Java" to each query, since these questions were performed on a code search tool that is specifically focused on retrieving Java code snippets extracted from Stack Overflow.

All code snippets were collected in May 2025 using the free versions of ChatGPT 4o, Gemini 2.5 Flash, and DeepSeek V3, with 327 snippets extracted from each model, totaling 981. Each generated Java class was extracted from a separate Java file. In cases where the LLMs initially responded with pseudocode or step-by-step explanations before providing a full Java class, only the final code was retained, and any preliminary pseudocode was discarded. If the solution involved multiple classes, they were placed in the same file with only one declared public, typically the main class.

2.2. Selecting Readability Rules from Linters

The code snippets extracted in the previous section were evaluated using the SonarLint tool [SonarLint 2025], with the aim of identifying potentially code readability warnings. This tool can detect readability warnings based on the rules defined in its documentation [SonarLint 2025] and developers generally recognize the relevance of Static Analysis Tools (ASAT) for overall software improvement [Marcilio et al. 2019, Romano et al. 2022].

To extract the code readability warnings identified by SonarLint, a standalone client script was developed that integrates with the SonarLint core (version 10.22) available online. The script processes each code snippet collected from the LLMs by submitting it to SonarLint and retrieving the list of detected warnings. In total, SonarLint reported 3,694 warnings in 77 unique rules classified as code smells, as shown in step 3 of Figure 1. SonarLint warnings related to bugs or vulnerabilities were discarded.

Table 1 presents an example of the warnings generated by SonarLint and how they are used to extract the corresponding rules. For the query with ID = 26: "odbc connection to mysql in Java", SonarLint identified two readability warnings in the Gemini-generated code snippet: *java:S4925* on line 12 and *java:S1220* on line 1. During this process, the number of affected code snippets and the associated warnings are recorded for each rule key.

Of the 77 unique rules classified as code smells by SonarLint, each rule was clas-

Table 1. SonarLint recommended modifications on a Gemini code snippet

Rule Key	Start Line	End Line	Message
java:S4925	12	12	Remove this ”Class.forName()”, it is useless.
java:S1220	1	1	Move this file to a named package

sified into one of two classes: readability or non-readability, as illustrated in Step 4 of Figure 1. This classification is necessary because some code smells could affect other aspects, such as performance. This classification was independently performed by two evaluators. During the agreement process, disagreements occurred in only four of the 77 SonarLint rules, which were resolved through discussion among evaluators, supported by the SonarLint documentation, which contains descriptions, code examples, and tags for each rule. For example, seven rules labeled *performance* by SonarLint were discarded. As a result, 54 SonarLint rules addressing readability-related warnings were identified, resulting in 610 readability warnings reported by SonarLint. In this process, 23 rules were discarded.

2.3. Prompting LLMs for Code Readability Improvements

Among the 981 code snippets generated (327 by each model), 298 (30%) have readability warnings detected by SonarLint. This includes 104 Gemini, 80 ChatGPT, and 114 DeepSeek. Each of these 298 snippets was then provided as input to the three LLMs, with the aim of correcting the identified warnings. As a result, 894 new code snippets were generated, 298 by each LLM.

Figure 2 illustrates this process. In the query with ID = 26, mentioned in Table 1, ChatGPT and DeepSeek did not produce any readability warnings. However, Gemini generated the readability warning *java:S4925*. Then, this code snippet was submitted to the three LLMs in an attempt to correct the warning. DeepSeek returned a new snippet without any readability warnings. However, ChatGPT and Gemini removed the original *java:S4925* warning but introduced a new one, *java:S3457*.

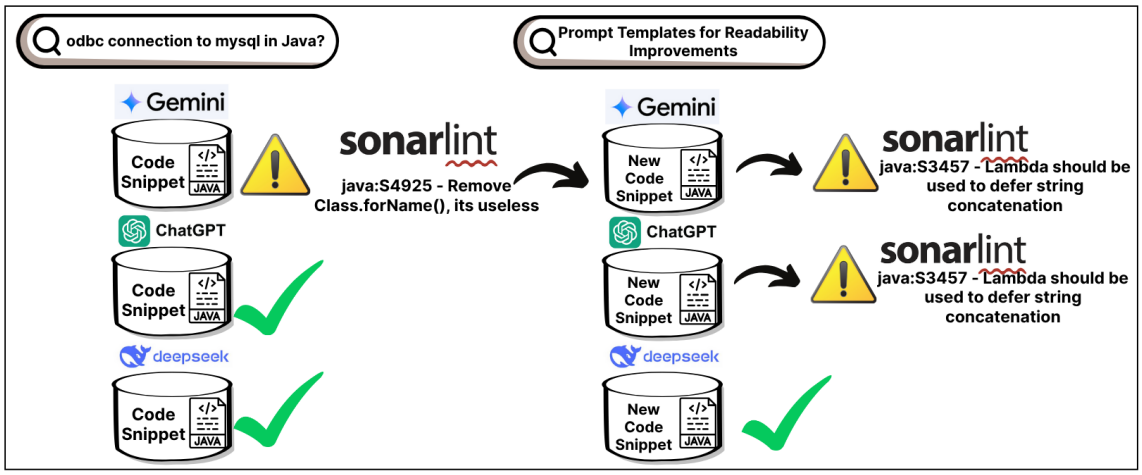

Figure 2. Improving Code Readability Example on Query ID = 26 “Odbc connection to Mysql in Java”

The prompt used to ask the LLM to improve code readability was created us-

ing a context defined through the system prompt, as shown in the text box below. In this prompt, `[CODE_SNIPPET]` represents the code snippet to be improved, and `[SONARLINT_WARNINGS]` contains the warnings raised by SonarLint.

Prompt Templates used for readability improvements

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."

The process was performed manually, simulating a typical developer interaction with the LLM.

3. Results

This section presents the results according to each previously defined research question.

3.1. Code Readability Warnings detected by Linters

Table 2 shows the number of warnings raised by SonarLint for each LLM, ordered by the total number of warnings. Due to space limitations, only rules with 10 or more warnings are shown. In general, ChatGPT produced the lowest number of code snippets affected (80 of 327 - 24%) and also the lowest number of warnings (151). Gemini had 104 (32%) affected code snippets and 198 warnings, while DeepSeek had 114 (35%) affected snippets with a total of 261 warnings.

To determine whether there are significant differences in the number of warnings per code snippet between LLM, the Friedman test was applied with a confidence level 95% (p-value < 0.05). The post hoc Nemenyi tests were then used to identify specific pairwise differences. The results indicate that ChatGPT has a significantly lower number of warnings compared to DeepSeek. However, no statistically significant differences were found between ChatGPT and Gemini under the same threshold.

There are four SonarLint rules focused on removing unused code: *java:S1481*, *java:S1854*, *java:S125*, and *java:S1068*. Among the models evaluated, DeepSeek performed the worst in terms of generating unused local variables and assignments. In contrast, ChatGPT was more prone to creating unused private fields, primarily due to the absence of the `getters()` and `setters()` methods.

For example, in query ID = 202: *How to access private field of different class in Java?*, both DeepSeek and Gemini generated a class with a private field accessed via public `getters()` and `setters()`. In the main class, Gemini invoked the `getter()` and printed the result, while DeepSeek called the `getter()` but ignored the returned value, raising warnings for an unused local variable and assignment. Interestingly, ChatGPT took a different approach: instead of creating accessors, it used reflection to access the private field through the `java.lang.reflect.Field` class. As a result, although the code was functional, the absence of `getters()` and `setters()` led SonarLint to report the private field as unused.

Table 2. Number of readability warnings and affected code snippets per SonarLint rule and LLM (*Snip.*: number of code snippets affected; *War.*: total number of warnings produced by SonarLint).

Rule	Description	ChatGPT		DeepSeek		Gemini	
		Snip.	War.	Snip.	War.	Snip.	War.
java:S1481	Unused local variables should be removed	31	39	58	88	20	28
java:S1854	Unused assignments should be removed	30	37	61	90	16	22
java:S125	Sections of code should not be commented out	4	4	9	10	28	31
java:S1192	String literals should not be duplicated	4	5	5	6	17	24
java:S1068	Unused ”private” fields should be removed	7	13	6	10	2	4
java:S3252	”static” base class members should not be accessed via derived types	6	6	5	5	7	7
java:S112	Generic exceptions should never be thrown	2	3	2	4	6	8
java:S1161	”@Override” should be used on overriding and implementing methods	6	8	4	5	0	0
java:S1602	Lambdas containing only one statement should not nest this statement in a block	4	4	6	8	0	0
java:S1604	Anonymous inner classes containing only one method should become lambdas	2	2	0	0	6	8
java:S117	Local variable and method names should comply with a naming convention	1	1	0	0	3	9
java:S3012	Arrays and lists should not be copied using loops	3	3	0	0	6	7
...	...	..	..	..	..	..	..
Total		80	151	114	261	104	198

Despite producing fewer unused variables and assignments, Gemini introduced significantly more commented-out code than the other LLMs. This occurred because while ChatGPT and DeepSeek provided alternative solutions in separate classes, Gemini included multiple versions within the same class using commented code blocks.

There are five SonarLint rules related to clarifying code intent: *java:S1192*, *java:S3252*, *java:S112*, *java:S1161*, and *java:S117*. Interestingly, Gemini performed the worst in most of these rules, except for *java:S1161*, where ChatGPT performed poorly by omitting the `@Override` annotation, which is recommended to clearly indicate the override of the method.

A notable example is query *ID = 16: How to create a thread in Java?*. All models produced a class that extends `java.lang.Thread` and overrides the `run()` method. However, only Gemini included the `@Override` annotation to make the override explicit. Despite this, Gemini introduced numerous `System.out.println` statements using the same string pattern `"Thread " + Thread.currentThread().getId()`, which raised warnings of the rule against duplicating string literals (*java:S1192*).

The remaining three warnings are associated with the preference for concise code (*java:S1602*, *java:S1604*, *java:S3012*). Gemini produced the least concise code, using anonymous inner classes instead of lambdas and manually copying loop logic. DeepSeek tended to generate lambdas with unnecessarily nested statements. An illustrative example is query *ID = 274: How do I press a button on the Java Swing when I press enter?*. As shown in the Listing 1, Gemini generated a verbose solution using an anonymous inner class. In contrast, DeepSeek and ChatGPT used lambdas to reduce boilerplate code, although both introduced unnecessary nesting in otherwise simple statements.

```
//Code Snippet Generated by Gemini
JButton okButton = new JButton("OK");
okButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(DefaultButtonExample.this,
            "OK Button Pressed!");
    }
});
//Code Snippet Generated by DeepSeek
JButton button = new JButton("Submit");
button.addActionListener(e -> {
    System.out.println("Submitted: " + textField.getText());
});
//Code Snippet Generated by ChatGPT
JButton button = new JButton("Click Me");
button.addActionListener(e -> {
    System.out.println("Button clicked!");
});
```

Listing 1. Gemini, DeepSeek and ChatGPT Code Snippets for a button event

RQ_1 Answer: ChatGPT produces code snippets with significantly fewer SonarLint warnings compared to DeepSeek. DeepSeek also generally generates more unused code, while Gemini tends to produce code that lacks clarity and requires better self-explanation. Both DeepSeek and Gemini also generate more redundant code overall. Although ChatGPT also violates some rules, it shows a more balanced performance across all categories.

3.2. LLMs Improving Code Readability

Table 3 presents the results of the LLM-based correction process applied to the 298 code snippets affected by code readability warnings. To analyze the table, consider the rule *java:S1481*, which was raised warnings in 109 of the 298 affected code snippets. After submitting these snippets to the LLMs for readability improvement, the warning was no longer present in 94 snippets modified by ChatGPT, 98 by DeepSeek, and 79 by Gemini. However, the warning remained uncorrected in 15 snippets from ChatGPT, 11 from DeepSeek, and 30 from Gemini. Additionally, the warning was introduced in 3 other snippets by ChatGPT, 4 by DeepSeek, and 2 by Gemini. The "Total" represents the number of code snippets, out of the 298 affected, for which ChatGPT, Gemini, and DeepSeek corrected, did not correct or introduced readability warnings. For example, ChatGPT corrected 226 of 298 snippets (76%), failed to correct 30 snippets (10%), and introduced new readability warnings in 42 snippets (14%).

Table 3. Number of code snippets by status (Corrected, Not Corrected, Introduced) for each SonarLint rule and LLM

Rule	Corrected			Not Corrected			Introduced		
	GPT	Deep	Gemini	GPT	Deep	Gemini	GPT	Deep	Gemini
java:S1481	94	98	79	15	11	30	3	4	2
java:S1854	94	97	76	13	10	31	3	4	2
java:S125	38	41	36	3	0	5	9	8	38
java:S1192	24	23	21	2	3	5	0	0	2
java:S112	16	17	17	1	0	0	3	1	0
java:S3457	3	4	4	1	0	0	19	2	16
java:S1068	14	15	15	1	0	0	3	8	5
java:S3252	18	18	18	0	0	0	0	0	0
java:S1602	10	10	10	0	0	0	0	0	1
java:S3012	9	8	8	0	0	1	0	0	1
java:S1604	7	8	8	1	0	0	0	0	1
...	..	..	..	..	..	..	..	..	..
Total	226	246	185	30	20	43	42	31	70

Although DeepSeek produced the highest number of readability warnings and affected code snippets, it also achieved the highest number of corrected code snippets compared to ChatGPT and Gemini. The fact that 35% of the affected code snippets were originally generated by DeepSeek (compared to 27% from ChatGPT and 28% from Gemini) may have influenced this result, as it was modifying its own code. However, ChatGPT showed similar performance, even though it was mostly correcting code snippets generated by other LLMs. Gemini had the lowest performance, affected by the frequent inclusion of commented-out code in its outputs, as observed in the rule *java:S125*.

Regarding the introduction of new readability warnings, the rule *java:S3457* recommends avoiding string concatenation using the "+" operator. However, both ChatGPT and Gemini introduced log statements that raised warnings about this rule, as seen in Query ID = 198: "create temporary file in Java". For example, ChatGPT added the line `logger.info("Temporary file created: " + tempFile.getAbsolutePath())`, which raised a warning related to string concatenation. In contrast, DeepSeek used the recommended approach with `LOGGER.log(Level.INFO, "Temporary file created: {0}", tempFile.getAbsolutePath())`, which avoids the warning. We also checked whether other SonarLint rules appeared in these new code snippets, but found only one occurrence, which appeared in a single instance.

In some of the cases where warnings were not corrected, particularly those involving unused variables or assignments, the LLMs often chose to remove the code entirely rather than just fix the warning. For example, in query ID 231 = "How to copy an array in Java", the original code generated by DeepSeek included the line `int[] copy = original.clone();`, but the variable `copy` was not used elsewhere in the code. After refactoring, Gemini removed this line, which led to a new issue in which the variable `original` was declared but never used. DeepSeek, on the other hand, changed the line to `original.clone();` to avoid the warning without introducing a new one.

An interesting case involves rules that were completely resolved by the LLMs in all affected instances. For example, the rule *java:S3252*, which recommends accessing the `EXIT_ON_CLOSE` constant through its interface `javax.swing.WindowConstants`, was consistently corrected. Another example is rule *java:S1602*, which suggests removing boilerplate code in lambda expressions.

RQ_2 Answer: DeepSeek corrected more code snippets affected by readability issues compared to the other LLMs, reaching 83% against 76% for ChatGPT and 62% for Gemini. It was also the LLM that introduced the fewest new readability warnings. However, manual analysis revealed that in some cases the LLMs failed to understand the context of the changes, occasionally removing code that should not have been removed.

4. Threats to Validity

In this study, the analysis of code readability is based solely on the warnings reported by SonarLint, which introduces an inherent limitation. Although SonarLint provides a large set of rules to detect potential readability problems , it may not capture all types of readability warnings present in the code. As a result, some relevant aspects of code readability could have been omitted.

A further threat is related to our interpretation of whether each SonarLint rule genuinely reflects a code readability concern. This study adopted a binary classification to filter the code smell rules that are concerned with improving code readability. To mitigate this threat, the rules were analyzed based on their tags and the classification in previous work [Dantas et al. 2023].

Finally, a construction threat refers to the manual process of extracting source code from LLMs, which is prone to human error. To mitigate this, we introduced validation

steps throughout each process. For example, if a code snippet contains errors that prevent its analysis, SonarLint throws an exception and stops execution. These validations were implemented to ensure that all code snippets were indeed analyzed.

5. Related Work

This section discusses four papers. The first two assess the code generated by LLM. The last two comprise the combination of static analysis tools with code generation.

Dantas et al. [Dantas et al. 2023] assessed the readability of the code generated by ChatGPT by comparing it with the human-written code snippets extracted from Stack Overflow. Their findings showed that ChatGPT produced fewer warnings related to code conventions, although it struggled with some of the newer Java features. This work differs by evaluating a broader set of LLMs including DeepSeek and Gemini, and by proposing an approach to automatically correct code snippets through LLM prompting.

Naser [Al Madi 2023] evaluated the readability of the code generated by GitHub Copilot in comparison to the code written by human developers, combining static analysis with eye tracking data. Although Copilot-generated code received less visual attention from programmers, suggesting greater readability, this work focused on a single tool. In contrast, our study compares multiple LLMs and introduces a method to automatically refactor code snippets by leveraging linter suggestions as input prompts to the LLMs.

Loriot et al. [Loriot et al. 2022] introduced Styler, an approach that addresses readability-related improvements such as code styling. Their method collects Checkstyle warnings for each class and trains a neural model to automatically apply the linter. This study has a similar objective on RQ_2 but focuses on SonarLint's readability-specific rules and evaluates how effectively each LLM (ChatGPT, DeepSeek, and Gemini) can refactor code snippets according to those suggestions.

Jaoua et al. [Jaoua et al. 2025] proposed a hybrid approach that combines linters with LLM to improve the quality of the code review process. Our approach differs by selecting a different linter and focusing specifically on code readability, rather than improving the precision of review comments.

6. Conclusion

In this study, three LLMs generated 981 code snippets in response to 327 queries. ChatGPT showed a more balanced behavior, producing snippets with fewer readability warnings detected by SonarLint and successfully removing warnings in 76% of the affected cases. DeepSeek generated more snippets with readability issues, but achieved the best performance when prompted to correct them. In contrast, Gemini had the lowest correction performance and introduced a higher number of new readability warnings.

Future work could assess whether the changes made by the LLMs preserve the original program behavior. This could be explored by selecting classes covered by unit tests. Other work could extract a combination of linters such as *Checkstyle* and *PMD* with SonarLint to produce a unified set of readability recommendations for LLM.

Acknowledgments

This work was supported by the BIC JR program of the Instituto Federal do Triângulo Mineiro (IFTM), which provided a research scholarship to one of the authors.

References

Al Madi, N. (2023). How readable is model-generated code? examining readability and visual inspection of github copilot. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

ChatGpt (2025). Chatgpt web page. `https://chatgpt.com/`. Accessed on May 10, 2025.

Crokage (2025). Crokage tool web page. `http://isel.ufu.br:9000/`. Accessed on May 10, 2025.

da Silva, R. F. G., Roy, C. K., Rahman, M. M., Schneider, K. A., Paixão, K. V. R., de Carvalho Dantas, C. E., and de Almeida Maia, M. (2020). CROKAGE: effective solution recommendation for programming tasks by leveraging crowd knowledge. *Empir. Softw. Eng.*, 25(6):4707–4758.

Dantas, C. E., Rocha, A. M., and Maia, M. A. (2023). Assessing the readability of chatgpt code snippet recommendations: A comparative study. In *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*, SBES '23, page 283–292, New York, NY, USA. Association for Computing Machinery.

DeepSeek (2025). Deepseek web page. `https://www.deepseek.com`. Accessed on February 6, 2025.

Fernandes, G., Maia, M. A., and Dantas, C. E. C. (2025). How Readable Is LLM-Generated Code Snippet? A Comparison of ChatGPT, DeepSeek, and Gemini - Replication Package. `https://doi.org/10.5281/zenodo.15803308`.

G1 Tecnologia (2025). App chinês deepseek supera chatgpt nos eua e derruba ações de empresas ligadas à ia. `https://encurtador.com.br/woy5r`. Accessed on February 6, 2025.

Google (2025). Gemini web page. `https://gemini.google.com/?hl=pt-BR`. Accessed on February 6, 2025.

Holmes, R., Cottrell, R., Walker, R. J., and Denzinger, J. (2009). The end-to-end use of source code examples: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*, pages 555–558.

Hora, A. (2021a). Googling for software development: What developers search for and what they find. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 317–328.

Hora, A. C. (2021b). Apisonar: Mining api usage examples. *Software: Practice and Experience*, 51:319 – 352.

Jaoua, I., Sghaier, O. B., and Sahraoui, H. (2025). Combining Large Language Models with Static Analyzers for Code Review Generation . In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 174–186, Los Alamitos, CA, USA. IEEE Computer Society.

Keivanloo, I., Rilling, J., and Zou, Y. (2014). Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 664–675, New York, NY, USA. Association for Computing Machinery.

Loriot, B., Madeiral, F., and Monperrus, M. (2022). Styler: learning formatting conventions to repair checkstyle violations. *Empirical Software Engineering*, 27.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., and Pinto, G. (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)*, pages 209–219.

Minelli, R., and, A. M., and Lanza, M. (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*, ICPC '15, page 25–35. IEEE Press.

Niu, H., Keivanloo, I., and Zou, Y. (2017). Learning to rank code examples for code search engines. *Empirical Softw. Engg.*, 22(1):259–291.

Piantadosi, V., Fierro, F., Scalabrino, S., Serebrenik, A., and Oliveto, R. (2020). How does code readability change during software evolution? *Empirical Software Engineering*, 25:1–39.

Romano, S., Zampetti, F., Baldassarre, M. T., Di Penta, M., and Scanniello, G. (2022). Do static analysis tools affect software quality when using test-driven development? In *Empirical Software Engineering and Measurement*, ESEM '22.

Sadowski, C., Stolee, K. T., and Elbaum, S. (2015). How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 191–201. Association for Computing Machinery, New York, NY, USA.

Silva, J., Dantas, C., and Maia, M. (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*, pages 125–136, Porto Alegre, RS, Brasil. SBC.

Sobania, D., Briesch, M., Hanna, C., and Petke, J. (2023). An Analysis of the Automatic Bug Fixing Performance of ChatGPT . In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30, Los Alamitos, CA, USA. IEEE Computer Society.

SonarLint (2025). Sonarlint web page. `https://www.sonarsource.com/products/sonarlint/`. Accessed on February 6, 2025.

Tufano, R., Mastropaolo, A., Pepe, F., Dabić, O., Penta, M. D., and Bavota, G. (2024). Unveiling ChatGPT's usage in open source projects: A mining-based study.

Zhang, Z., Xing, Z., Zhao, D., Xu, X., Zhu, L., and Lu, Q. (2024). Automated refactoring of non-idiomatic python code with pythonic idioms. *IEEE Transactions on Software Engineering*, PP:1–22.