Measuring how changes in code readability attributes affect code quality evaluation by Large Language Models

Igor Regis da Silva Simões
Banco do Brasil / Universidade de Brasília
Brasília, DF, Brazil
igor@bb.com.br

Elaine Venson
Universidade de Brasília
Brasília, DF, Brazil
elainevenson@unb.br

ABSTRACT

Code readability is one of the main aspects of code quality, influenced by various properties like identifier names, comments, code structure, and adherence to standards. However, measuring this attribute poses challenges in both industry and academia. While static analysis tools assess attributes such as code smells and comment percentage, code reviews introduce an element of subjectivity. This paper explores using Large Language Models (LLMs) to evaluate code quality attributes related to its readability in a standardized, reproducible, and consistent manner. We conducted a quasi-experiment study to measure the effects of code changes on Large Language Model (LLM)s' interpretation regarding its readability quality attribute. Nine LLMs were tested, undergoing three interventions: removing comments, replacing identifier names with obscure names, and refactoring to remove code smells. Each intervention involved 10 batch analyses per LLM, collecting data on response variability. We compared the results with a known reference model and tool. The results showed that all LLMs were sensitive to the interventions, with agreement with the reference classifier being high for the original and refactored code scenarios. However, this agreement diverged for the other two interventions. The LLMs demonstrated a strong semantic sensitivity that the reference model did not fully capture. A thematic analysis of the LLMs' reasoning confirmed their evaluations directly reflected the nature of each intervention. The models also exhibited response variability, with 9.37% to 14.58% of executions showing a standard deviation greater than zero, indicating response oscillation, though this did not always compromise the statistical significance of the results. LLMs demonstrated potential for evaluating semantic quality aspects, such as coherence between identifier names, comments, and documentation with code purpose. Further research is needed to compare these evaluations with human assessments and explore real-world application limitations, including cost factors.

KEYWORDS

Code Quality, Code Comprehensibility, Static Analysis, Software Engineering, LLM, ChatGPT, Gemini, Llama, Claude

1 Introduction

Reading source code is necessary for developers to understand the software's purpose, making it a central activity in software maintenance [1]. This activity of understanding the software or program can occur top-down or bottom-up. In the top-down approach, the developer starts by reading documents such as requirements and architectural documents, then moves to implementation layers. In the bottom-up approach, the developer starts directly from the code implementation [2].

Source code can be readable and comprehensible. These two characteristics are often treated as the same, but they are distinct textual features. Readability relates to the complexity of the presented text, while comprehensibility relates to the reader's ability to grasp the text's meaning. Thus, code measured as readable may still be incomprehensible to a reader lacking the necessary knowledge or experience to understand it [3].

This research focuses on two commonly studied readability attributes of source code - identifier names and code comments, as detailed in Section 3. We do not address other artifacts that comprise the software (documentation, architecture, etc.). Also, our study do not cover the code understandability research area, as it would require evaluating developers and their characteristics, such as seniority and familiarity with the language in which the code is written [4].

Low code readability is treated as a type of technical debt. Static code analysis tools aim to map violations of coding best practices to measure this debt. SonarQube classifies this type of violation as: *"Code Debt - Refers to problems found in the source code (violating best practices or coding rules) that negatively affect its readability and make it difficult to maintain."*[1] [5].

Low source code readability affects the industry, causing negative impacts over developers' productivity [6] and job satisfaction [7]. Many research studies have been conducted to advance the understanding of code readability metrics [8]. Several approaches have been developed, yet none of them reach the same level of accordance with human evaluation, showing that there is still gaps to be filled [9, 10, 11, 12, 13, 14].

The rise of LLMs and their emergent abilities, opens up new avenues for research. There are works investigating which code characteristics LLMs are sensitive to [15], how LLMs can be used to evaluate code quality [16], and how they can support personalized assessments of code readability for individual developers [17]. These preliminary results point to the need for a better understanding of how LLM assess code readability.

In this work, we explore these issues, addressing the following research questions:

(1) **Can LLMs be used to generate a metric that assesses code quality attributes associated with readability?**
(2) **How code attributes such as identifier naming and code comments affect readability and quality perception by an LLM?**
(3) **How can the inherent variability of LLMs affect their evaluation of code readability and quality?**

[1] SonarQube https://www.sonarsource.com/learn/technical-debt/#types-of-technical-debt

The remainder of this paper is organized as follows: Section 2, explores the studies addressing the researched topic. Section 3, details the method applied in this study. Section 4, presents the findings from each stage of the experiment. Section 5, provides a general analysis and discussion of the results from all experiment stages. Limitations and future work directions are presented in Section 6; and finally, the conclusion is in Section 7.

2 Related work

This work lies at the intersection of LLMs' and code readability and comprehensibility. We conducted a comprehensive review of literature exploring these intersections. Firstly, we examined automated approaches for measuring code readability and comprehensibility. Secondly, we investigated how LLMs are utilized to enhance code readability and comprehensibility. To conduct this review, we used the results of the systematic mappings from Bexell [18], Wyrich, Bogner, and Wagner [8], and Hou et al. [19].

Bexell [18] mapped 76 studies related to code readability, two of which specifically address the automation of its measurement.

Wyrich, Bogner, and Wagner [8] mapped 95 studies on code comprehensibility, two of which study its automation, with one being the same found in Bexell [18]'s mapping. Other studies have also conducted some form of automation related to source code quality [20, 10, 4].

Hou et al. [19] mapped 395 studies on the application of LLM in various software engineering tasks, eight of which were classified as "Code understanding" research. Three studies investigate or propose improvements in LLMs to enhance their performance in code-related activities, but without aiming to use LLMs for code comprehension [21, 22, 23]. A fourth study presents a dataset for training LLMs, demonstrating its effectiveness in improving models' capabilities in code generation and comprehension tasks [24].

We classified the mapped studies, focusing on automated approaches. The studies are grouped into three main categories: (1) Machine Learning Based Approaches and (2) Deep Learning Based Approaches.

Machine Learning Based Approaches: This group focuses on using quantifiable metrics extracted from the source code, documentation, and developer profiles to predict or explain code comprehensibility. These studies often employ statistical analysis and machine learning techniques to identify correlations and build predictive models. Buse and Weimer [9] introduced a metric for software readability using the average score provided by 120 human annotator to train their binary classifier. Dorn [10] also identified promising results, when compared with previous metric, on automating the measurement of code readability using also linguistic attributes (e.g., validating identifier names with English dictionaries). Further work also explored lexical artifacts [11]. Scalabrino et al. [25] demonstrated that no single code attribute can be used to explain code comprehensibility in isolation. Trockman et al. [26] developed a binary logistic regression model, reanalyzing the data from Scalabrino et al. [25], considering source code properties, documentation, and developer profile data (e.g., experience). This study highlighted the importance of combined metrics and revealed that code understandability would be measurable in future. Scalabrino et al. [4] investigated 121 metrics related to code, documentation, and developers. They found weak correlations between individual metrics and perceived comprehensibility. Although combining metrics showed some discriminatory power, the results were insufficient for practical use [12]. Additionally, those studies lack the evaluation of semantic aspects, such as meaning of identifier names and the meaning of the text in a comment, in relation to the code.

Deep Learning Based Approaches: This group utilizes Natural Language Processing (NLP) and other deep learning techniques to analyze code and related text (e.g., comments, documentation) to understand code semantics and improve comprehensibility. Kanade et al. [27] presented CuBERT (Code Understanding BERT), achieving superior results in various code-related tasks, with one related in correlating comments with code (docstring task). Shen et al. [28] benchmarked pre-trained models (including CuBERT and CodeBERT) for identifying syntactic structures in code. They found that these models still lacked comprehensive code syntax understanding, achieving results inferior to simpler reference models. This highlights the challenges of applying NLP models to capture the nuances of code syntax.

Mi et al. [13] introduced the use of Convolutional Neural Network (CNN) trained on a matrix representation of code, generated by its characters ASCII values, inspired by image recognition approaches. Their model was compared to previous ML based models and achieved a slightly improved accuracy compared to the best one, from Scalabrino et al. [11]. Latter Mi et al. [29] presented a CNN trained on code representation based on three levels of information: character-level (ASCI value), token-level (key word) and node level (Abstract Syntax Tree (AST)). Their model demonstrated improved compared to previous works on readability classification, including their previous model.

Seeking for improvements Mi et al. [14] presented a novel approach using CNN to extract structural features from the ASCII matrix representation of code. Also a combination of CNN and BERT model to generate token representation and extract semantic features from code. Finally they have used a CNN to extract visual features from RGB matrix generated from source code screen shots. Their new approach obtained the best performance compared to all previous, leading to the hypothesis that different deep learning technics combined can extract unmapped readability features. They also demonstrated that visual features contributed the least to the readability assessment, while the structural features being the most relevant.

LLM-Based Approaches: This group explores the use of LLMs for code comprehension and readability. Simões and Venson [16] investigated the use of LLMs for source code quality assessment, including readability, by comparing the results with SonarQube analysis. Their study shows the LLM potential for evaluating code quality, but indicates the need for further studies. Hu et al. [15] found that LLMs are highly sensitive to semantic perturbations in code. However, their work did not seek to develop a metric based on this property. Vitale et al. [17] evaluated the use of LLM to generate a personalized readability assessment, by using previous works' datasets [20, 10, 12] and adopting three labels "unreadable", "readable", "neutral". They compared their LLM approach with Scalabrino's Classifier, using precision, recall and f-1 score metrics. Their results demonstrated that the use of LLMs for personalized readability are less effective than state-of-art generalist model.

Studies related to code readability and comprehensibility have seen a progression from classifiers using machine learning algorithms [20] to NLP models [27], and more recently Deep Learning [30]. Current research focuses on evaluating the use of LLMs for evaluate code quality [16] and readability [17].

Our work falls on LLM-Based Approach. The findings in this work can be used to direct further investigation towards the development of a code readability metric, or even a more broad novel code quality metric.

3 Methodology

In this work, we study the response of a set of objects (nine LLMs) to the same set of interventions (code interventions) into the same set of subjects (12 Java Classes). The tested hypothesis consider that LLMs will change its evaluation of source code accordingly to the interventions. There is no random selection of interventions, objects, or subjects, thus we are conducting a quasi-experiment [31]. This is a type of experiment often used in software engineering research [32].

Therefore, we conducted a *"blocked subject-object"* study, as we have nine study objects (LLMs) and 12 subjects (12 classes) that will undergo changes (intervention) in their characteristics commonly related to readability and source code quality, which will be measured by the LLMs [32].

Fakhoury et al. [33] identified that tools used to detect style problems are able to capture readability characteristics of source code. In their study, those type of tools demonstrated better sensibility than models proposed by Scalabrino et al. [11] and Dorn [10]. The use of a comparison group is necessary to identify random changes on LLMs' metric that are not caused by source code changes, but inherent LLM's variability.

We included two approaches as comparison groups [31]. One approach is the static analysis software SonarQube and some of its metrics. We chose this tool because it is broadly used in industry and some of its metrics evaluate code style problems. The other comparison reference is the Scalabrino et al. [12] model, which combine several source code features and had its accuracy extensively evaluated in previous works[33, 4, 34]. Their reproducibility toolkit[2] has a working software, able to automatically extract features and perform code readability classification, without human intervention. This characteristic ensures that we do not introduce interpretation errors when building our own version of their model.

Yet, according to Shadish, Cook, and Campbell [31], the use of control groups are of minimal advantage unless they are also accompanied by pretest measurements taken on the same outcome variable as the posttest. So we executed a pretest, i.e, all source code was evaluated by SonarQube, Scalabrino Classifier and LLMs' before any code intervention, mitigating selection bias for both, source code and LLMs', creating a baseline metric.

The submission of the classes and their interventions to the LLMs' analyses was not randomized. The interventions conducted on code were not random since they were guided by the measurements of the control measurement tool. Also, it was not necessary any randomized LLM analysis, since the LLM's had no context about previous code intervention, being unable to retain this information and compare the different states of code, which could affect its evaluation.

The LLM's were instructed to output its evaluation similarly to a binary classification, i.e. emit a 0 to 100 score. This approach ensures a direct comparison with the Scalabrino Classifier. Details about this approach are in Section 3.3.

3.1 Comparison Group - SonarQube Metrics

SonarQube has a set of rules known as Code Smells. According to its documentation, Code Smells are issues related to maintainability.

Currently, SonarQube is replacing this definition with a more granular one called Clean Code, which includes the following attributes: consistency, intentionality, adaptability, and responsibility. The first two attributes relate to code readability characteristics.

According to SonarQube's documentation "consistent code is formatted, conventional, and identifiable" and "intentional code is clear, logical, complete, and efficient".

SonarQube also measures the percentage of documentation lines relative to the code lines of a class, as well as the total number of lines in the class.

In this experiment, we manipulate code attributes that affect metrics related to its readability, which can be captured by SonarQube's Clean Code metrics and documentation-related metrics directly measured by the percentage of documentation.

To use SonarQube as our comparison group for metrics, we utilize Clean Code metrics and their attributes: Consistency (C), Intentionality (I), Adaptability (A) and Responsibility (R) representing the number of issues detected into the analyzed source code. Higher values for (C) and (I) are supposed to reduce source code readability. The percentage of comments or documentation (D), and the total number of lines (L) in each manipulated class as the SonarQube's documentation metric is a ration of the total lines of code. We refer to all Clean Code violations as Code Smells.

3.2 Comparison Group - Scalabrino's Classifier

As an approach evaluated against several datasets and also compared against code style tools, with an outstanding performance, we will take Scalabrino's classifier as our ground truth metric. It was tested against 600 code snippets, evaluated by 5k+ people.

The classifier provided by Scalabrino et al. [12] works as a command line tool that receives either a code snippet with a Java method, an entire class file or a folder with any of these types. Upon its execution it produces a score between 0 (surely unreadable) and 1 (surely readable), rating the code readability.

The provided tool uses a Opens Source Software (OSS) library to parse an AST. As its last update was in 2021, in order to make the tool work with current Java syntax, we updated the AST parser library. The model combines structural and textual features. Comments Readability, Textual Coherence and Number of Concepts are the three most important textual features for this model. That makes it as the right fit to evaluate the type of code interventions we have performed, further explained in section 3.5.

The scores obtained with this classifier will be named as Scalabrino Classifier (SC). The classifier's output is a number with 16 digits

[2]https://dibt.unimol.it/report/readability/

precision, which we convert to the same LLM scale (0 to 100), with one digit precision.

3.3 Objects - Selection of LLMs

We studied nine objects, which are the LLMs: ChatGPT (4o and 4o-mini), Gemini (2.0 Pro and Flash), Llama (3.1 405B and 3.1 8B), Claude (3.7 Sonnet and 3.5 Haiku) and Deep Seek V3. Each of the first four listed LLMs has both a robust and a fast version, except Deep Seek V3.

The objective is to evaluate the capability and sensitivity of the LLMs in detecting changes in code attributes that affect its readability, compared to the detection made by the control tool, SonarQube.

There are various benchmarks for LLMs, covering a wide range of applications and tasks, from general to more specific purposes [35]. This study does not aim to create or conduct a benchmark but rather to evaluate the performance of these models in assessing code readability.

However, given the constant evolution of commercial and open-source models, selecting models for research use becomes challenging. To select the best and most current models, we referred to the ranking on the Artificial Analysis website[3], which classifies models offered through rest API, based on three main comparisons: Intelligence, Speed, and Price.

In this work, we did not analyze costs, so we selected the top four models in quality and the top four models in speed. If a model appeared in more than one criterion, we selected the next model in the ranking where the duplicated model had the least affinity.

At the time of planning this work, the top four models in quality and speed are the ones listed in Table 1. We did not considered the models classified as reasoning model. The ChatGPT 4o-mini model was also ranked second by quality, but since its purpose is speed and it is present in the respective ranking, we removed it from the intelligence list. Similarly Gemini 2.0 Pro ranked third on speed, but we considered it on intelligence rank only.

During execution of this research (between 10/2024 and 03/2025), some models received updates and the Deep Seek V3model entered the rank. We decided to execute the update and to add this new model.

Table 1: Selected models

Model	Score	Release Date	Criteria
Gemini 2.0 Pro	49	05/02/2025	intelligence
Claude 3.7 Sonnet	48	19/02/2025	intelligence
Deep Seek V3	46	26/12/2024	intelligence
ChatGPT 4o	41	13/05/2024	intelligence
Llama 3.1-405B	40	23/07/2024	intelligence
Gemini 2.0 Flash	265	05/02/2025	speed
Llama 3.1-8B	185	23/07/2024	speed
ChatGPT 4o-mini	79	18/07/2024	speed
Claude 3.5 Haiku	66	13/03/2024	speed

All models were used through REST APIs from the following sites: OpenAI[4], Google[5], Anthropic[6], DeepInfra[7] and DeepSeek[8].

[3]Artificial Analysis https://artificialanalysis.ai/
[4]OpenAI https://openai.com/
[5]Google AI Studio https://ai.google.dev/aistudio
[6]Anthropic https://www.anthropic.com/
[7]DeepInfra https://deepinfra.com/
[8]DeepSeek https://deepseek.com

These models are able to learn from a few examples. This capability is due to various factors, including their number of parameters, computational power, and the dataset used in their training. These Few-Shot Learning (FSL) abilities, referred to as *emergent abilities* by Wei et al. [36], have enabled the software to perform tasks beyond its original design. Furthermore, new LLMs have become capable of surpassing results considered the state-of-the-art for specialized models [36]. In this work, we adopt a Zero-Shot Learning (ZSL) approach.

To create the prompt that guides the LLM on how to respond to the assigned task, we used three patterns mapped by White et al. [37] (*Persona, Template, and Reflection*). We instructed the LLM to assume a persona capable of performing the analysis with a certain knowledge. For this purpose, we used the passage: *"The assistant is a seasoned senior software engineer, with deep Java Language expertise, doing source code evaluation as part of a due diligence process, these source codes are presented in the form of a Java Class File. Your task is to emit a score from 0 to 100 based on the readability level of the source code presented."*.

The LLM was requested to present an explanation for the grade, by passing guidelines on its format *"- The "explanation" attribute must not surpass 450 characters and MUST NOT contain special characters or new lines."*. To enable the LLM's responses to be processed via software, we requested it to follow a template for the response: *"Your answers MUST be presented ONLY in the following json format: "score":"NN%", reasoning:"your explanation about the score""*. This explanation was used for debug purposes to iterate between prompts improvements.

In all executions, the prompt remained unchanged. Each class was analyzed 10 times, totaling 120 analyses and consuming about 1.2 million tokens per model on each scenario. The total number of analysis for this research was 4.320 or around 43.2 million tokens. The results were submitted to an analysis of the variability of the responses presented by each LLM. The output files of all executions, as well as the Python code used to perform these analyses, are preserved in a GitHub repository [38].

3.4 Subjects - Algorithms for analysis

The most commonly used language in studies related to code readability and comprehensibility is Java, with 47% of published articles. Over 40 years of research, more than 88% of studies have used only one programming language in their investigations. The most common approaches for obtaining code snippets for readability studies were: (1) code produced by researchers, (2) code from open-source projects, and (3) code used in previous works [8]. Also, Scalabrino et al. [12] model has proven results only against Java code, as well as its toolkit is able to extract features only from Java AST.

For such reason, we decided to use Java language software in this research and selected the open-source Open JDK project[9], which refers to the fundamental libraries of the Java SDK. We selected two classes with rich API documentation, with the intention to clearly detect the impact of a rich documentation removal from the sour code. The two selected classes were *java.util.DoubleSummaryStatistics* and *java.time.Month*.

[9]Open JDK https://github.com/openjdk/jdk

Additionally, a set of demonstration classes from the JDK was selected. This set of classes was chosen for their less rich documentation since they were written to demonstrate the language's use. The list of selected classes is as follows: root package ../demo/share/jfc/, sub packages SampleTree, classes DynamicTreeNode, SampleData, SampleTree, SampleTreeCellRenderer, SampleTreeModel, sub package Notepad, classes ElementTreePanel, Notepad, sub package Stylepad, classes HelloWorld, Stylepad and Wonderland.

The total number of original selected classes is 12, totaling 4,051 lines. Each class was used in 4 different intervention states, totaling 48 classes with 14.919 lines. The number of classes and code was kept low to allow for manual interventions in the code. The main studies in this area have used the following datasets: Buse and Weimer [20] dataset with 100 snippets totaling 769 lines of code, Dorn [10] has 120 snippets totaling 3,617 lines and Scalabrino et al. [12] has 200 snippets totaling 5.140 lines. Sergeyuk et al. [34] has 120 snippets totaling 5,533 lines.

3.5 Interventions in the algorithms

To explore how source code attributes affect the scores assigned by the LLMs, we perform interventions on the code, followed by re-analyses by SonarQube and the LLMs. After each intervention, we present the results from the LLMs. **Identifier naming** and **code comments** are the two most studied source code attributes related to code readability and comprehensibility [8]. The interventions will target these two attributes. We did not test if the changes cause an improvement or deterioration to readability, other than compare the results with the evaluation of comparison group.

Intervention on code comments: In the second analysis, we identify the impact of comments on SonarQube and LLM evaluations by removing the comments and re-running the analysis. All comments were removed except source code that was commented out.

Intervention on variable names: In the third analysis, we alter the names of variables and methods. Good variable and method names facilitate code comprehension. This is a widely accepted axiom by industry [39] and extensively investigated showing that consistent use of descriptive names improve code readability [8, 40, 41, 11, 20].

In order to emulate such an extreme intervention as the removal of all comments, we modify all variables and methods to semantically confusing values, such as names of vegetables and spices (potato, tomato, parsley, rosemary etc).

We expect SonarQube not to show changes in its evaluation, but the LLMs should be able to identify the confusion caused in the code, impacting the score. Class names are preserved to allow comparison of results, as well as overridden methods.

Intervention on Code Smells: In the fourth analysis, we fix all fixable Code Smells, especially those of Clean Code Attributes C and I, that are strongly related to readability.

We expect these changes to positively impact SonarQube metrics as well as the LLMs' evaluations. Changes are performed only on methods and classes containing a Code Smell. Thus, classes C_1, C_7 and C_{10}, which do not have Code Smells, do not undergo intervention at this stage, as can be seen on Table 2, detailed on Section 4.1.

Code versions: The interventions generate four code versions for the 12 classes (with the identifier we use to refer to each scenario in parentheses):

(OC) Original code state;
(I1) without comments;
(I2) with confusing names for identifiers;
(I3) with fixed Code Smells.

In Section 5, we compare the results of all analyses.

Every code analysis involved statistical evaluation of the LLM performance. For each scenario ($S \in \{OC, I1, I2, I3\}$) and Java class (C_i, where $i \in \{0, 1, \ldots, 12\}$), we calculated the mean and standard deviation of the scores from 10 analysis samples for each LLM.

To compare LLM performance across scenarios for each class, we computed the precision, recall and f-1 score between LLM scores and the Scalabrino Classifier (SCO), taken as our ground truth, as previous work [17]. To stablish this ground truth, we have applied the same interval found by Piantadosi et al. [42] for the SCO. In their work they have found that the readability prediction tool is unreliable when its output score is within the range of [0.416, 0.600]. Based on this, we established thresholds for classification: a score below 0.416 classifies the code as 'unreadable,' and a score above 0.600 classifies it as 'readable.' Scores within this interval are considered 'neutral'. We applied the same thresholds to all LLM's scores.

3.6 Thematic Coding of LLM Reasoning

To systematically analyze the LLMs reasoning for the scores they assigned, we employed an inductive thematic analysis methodology. This approach, as outlined by Guest, MacQueen, and Namey [43], is designed to identify and examine themes from textual data in a transparent, credible, and systematic manner. The primary objective of this analysis was to understand the qualitative aspects and the criteria the LLMs used to evaluate code readability across the different intervention scenarios. The process was executed as follows:

Data Preparation and Segmentation: Comments from all LLMs for a given scenario have been consolidated into a single file, containing one comment per line.

Inductive Theme Identification: We did not start with a preconceived set of themes. Instead, two researchers independently read a subset of the LLM reasoning texts. The goal was to become familiar with the data and to notice recurring concepts, ideas, and justifications—the emergent themes.

Codebook Development: After the initial pass, a preliminary list of themes was discussed and refined. This led to the creation of a formal codebook. For each code, we developed **A descriptive label:** A short mnemonic for easy reference (e.g., `Good Readability`) and **Inclusion/Exclusion Criteria:** A set of keywords or short expressions to be used for regular expression matching.

Coding and Inter-Coder Agreement: For each thematic code in the codebook, a corresponding set of regular expression patterns was developed to identify its presence within the textual data. To avoid false positives in cases of ambiguous explanations, whenever a negative code (e.g., `Poor Readability`) was detected, the corresponding positive code (e.g., `Good Readability`) was not accounted for in the same comment.

Data Reduction and Analysis: The final step was to analyze the resulting structured data. This was achieved by quantifying the frequencies of each thematic code for each of the four scenarios (OC, I1, I2, and I3).

This structured and inductive approach ensured that the analysis of the LLMs' reasoning was not merely a subjective interpretation but a methodologically sound process, grounded in the data itself and aligned with established practices in qualitative data analysis.

4 Results

4.1 Analysis of the original code with comparison group

The metrics shown in Table 2 were collected to be used as the basis for comparison with the LLMs' results. A total of nine Code Smells were identified by SonarQube. All classes were rated A for maintainability, indicating a high presumed quality level. All classes are referenced in this section by its alias, as listed in table 2.

Table 2: SonarQube attributes values

Alias (class name)	Clean Code				D	L	SC
	C	I	A	R			
C_1 (DoubleSummaryStatistics)	-	-	-	-	62.9%	294	90.7
C_2 (Month)	2	1	-	-	63.4%	526	91.0
C_3 (DynamicTreeNode)	-	2	-	-	32.1%	155	93.3
C_4 (ElementTreePanel)	-	3	2	-	23.0%	579	78.9
C_5 (HelloWorld)	-	3	2	-	11.0%	177	64.4
C_6 (Notepad)	9	20	5	-	10.2%	824	77.4
C_7 (SampleData)	-	-	-	-	25.5%	75	93.3
C_8 (SampleTree)	-	5	5	-	27.4%	596	74.8
C_9 (SampleTreeCellRenderer)	-	1	1	-	16.5%	131	80.2
C_{10} (SampleTreeModel)	-	-	-	-	38.2%	48	89.1
C_{11} (Stylepad)	-	15	7	-	7.9%	378	74.0
C_{12} (Wonderland)	-	1	6	-	5.7%	268	68.5

Only three of the 12 classes do not have any Code Smell. Most of the Code Smells are of type C and I, strongly related to source code readability according to SonarQube documentation. The C_1 and C_2 classes have more than 60% of comments, and all classes have some level of comments. The complexity indicators show that some of the classes have a low level of complexity while others have higher levels, yet still considered low.

4.2 Analysis of the original code with the LLMs

Next, we list the models that presented variation on score attribution, with their respective standard deviation (SD) and coefficient of variation (CV), using notation C_i SD (CV):

ChatGPT 4o - C_7 1.6 (1,67%)
ChatGPT 4o-mini - C_4 4.2 (5,08%)
Llama 3.1-405B - C_4 2.4 (2,89%)
Gemini 2.0 Pro - C_4 4.8 (6.62%), C_6 1.6 (2.24%), C_9 2.6 (3.01%) and C_{11} 2.6 (3.59%).

Half of the models did not show any variation in the scores assigned.

The C_1 and C_2 classes, which are rich in documentation, received the highest evaluations from the LLMs.

As shown in Table 3, all nine evaluated LLMs demonstrated 100% agreement with the Scalabrino Classifier (SC) for the original code scenario. In this scenario, all classes were classified as readable by SCO and by all LLM.

Table 3: Agreement with Scalabrino Classifier - Original Code

	ChatGPT		Gemini		Llama		Claude		Deep
Metric	4o	4o mini	2.0 Pro	2.0 Flash	3.1 405B	3.1 8B	3.7 Sonnet	3.5 Haiku	**Seek** V3
Precision	100%	100%	100%	100%	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%	100%	100%	100%	100%
F1-Score	100%	100%	100%	100%	100%	100%	100%	100%	100%

4.3 Analysis after removing comments

In the second analysis, Sonar's rating remained A, but the number of Code Smells increased due to the repercussion of removing comments, as classes with public methods lacking JavaDoc, violating a quality rule[10], or empty blocks of code, that should at least be documented[11].

In Table 4, we marked the measures reductions in red and increments in green. The only Code Smell reduction was due to the removal of a comment with a FIXME string that was generating a Code Smell by SonarQube, indicating the need to developer to take an action[12]. As expected, the total lines of code (L) and documentation ratio (D) decreased, except for the C_5, C_8, C_{11} classes, which contain commented-out source code that was kept after the intervention. The SC also decreased the score, but not for class C_5.

Table 4: SonarQube values after comments removal

Class	Clean Code				D	L	SC
	C	I	A	R			
C_1	-	1	-	-	0.0%	99	80.0
C_2	2	1	-	-	0.0%	177	77.1
C_3	-	2	-	-	0.0%	103	82.1
C_4	-	3	2	-	0.0%	424	59.7
C_5	-	4	2	-	1.4%	166	71.5
C_6	9	19	5	-	0.0%	699	71.1
C_7	-	-	-	-	0.0%	47	87.7
C_8	-	5	5	-	5.0%	451	61.7
C_9	-	1	1	-	0.0%	109	47.1
C_{10}	-	-	-	-	0.0%	28	69.4
C_{11}	-	15	7	-	1.3%	239	70.5
C_{12}	-	2	6	-	0.0%	69	61.9

Below, we list models that presented variation on score attribution in this scenario, along with their respective SD and CV:

Claude 3.7 Sonnet - C_{10} 5.2 (6.54%);
Claude 3.5 Haiku - C_1 2.6 (2.97%);
Llama 3.1-8B - C_4 6.3 (8.11%);
Llama 3.1-405B - C_1 2.2 (2.58%) and C_7 2.6 (3.11%);
Deep Seek V3 - C_1 2.6 (2.85%) and C_7 2.1 (3.32%);
Gemini 2.0 Pro - C_5 3.2 (4.16%), C_8 1.6 (2.28%), C_{10} 2.6 (3.01%), C_{11} 2.1 (2.97%) and C_{12} 1.3 (1.63%).

Table 5 lists the variation between the results of the first and second analyses. The new distribution of LLM scores showed 38 score reductions and nine score increases. We marked the reductions in red, increases in green, changes with moderate statistical significance in yellow and with no statistical significance in gray.

All models presented at least one score variation, ChatGPT 4o showed only one reduction, while Deep Seek V3 had two - one with no statistical significance and the other with moderate significance. Claude 3.7 Sonnet presented variations for nine classes, all statistically significant. Seven out of nine changes pointed by Gemini 2.0

[10]SonarQube Rule: https://rules.sonarsource.com/java/RSPEC-1176/
[11]SonarQube Rule: https://rules.sonarsource.com/java/RSPEC-108/
[12]SonarQube Rule: https://rules.sonarsource.com/java/RSPEC-1134/

Table 5: Average score by class and LLM - Comments removed

	ChatGPT		Gemini		Llama		Claude		DeepSeek	SC
Class	4o	4o-mini	2.0-Pro	2.0-Flash	3.1-405B	3.1-8B	3.7-Sonnet	3-Haiku	V3	
C_1	0.0%	0%	-5.3%	-5.6%	-9.8%	-13.0%	-5.6%	-8.4%	-2.6	-11.82%
C_2	0.0%	-10.5%	0.0%	0.0%	0.0%	0.0%	-5.3%	0.0%	0.0	-15.24%
C_3	0.0%	-11.8%	0.0%	-11.8%	-15.3%	-34.8%	-11.8%	-23.5%	0.0	-12.00%
C_4	0.0%	-9.6%	-4.1%	0.0%	-4.2%	-8.2%	-11.8%	0.0%	0.0	-24.33%
C_5	0.0%	0.0%	1.3%	0.0%	0.0%	-25.0%	0.0%	13.3%	0.0	11.00%
C_6	0.0%	0.0%	-0.7%	0.0%	16.7%	0.0%	-11.8%	0.0%	0.0	-8.14%
C_7	-10.1%	0.0%	0.0%	-5.3%	-2.4%	0.0%	-11.8%	-11.8%	-4.2	-6.00%
C_8	0.0%	-11.8%	0.7%	0.0%	-12.5%	0.0%	-11.8%	-11.8%	0.0	-17.44%
C_9	0.0%	0.0%	-2.9%	0.0	-5.9%	0.0%	0.0%	-11.8%	0.0	-41.26%
C_{10}	0.0%	0.0%	2.9%	-5.6%	0.0%	-5.9%	5.3%	0.0%	0.0	-22.01%
C_{11}	0.0%	0.0%	-1.4%	0.0%	0.0%	-25.0%	0.0%	0.0%	0.0	-4.72%
C_{12}	0.0%	0.0%	3.2%	0.0%	14.3%	0.0%	15.4%	0.0%	0.0	-9.66%

Pro had no statistical significance, making it the model with the most noise.

Table 6 shows a uniform recall of 83.3% across all nine LLMs, as they have successfully identified most of the classes in accordance with SC. The precision is lower than the recall. This suggests that the LLMs incorrectly classified a number of unreadable classes (according to the SC) as readable, resulting in a higher rate of false positives. This was caused by 2 of the 12 classes being on neutral range by Scalabrino Classifier, while all classes remained classified as readable by all LLM.

Table 6: Agreement with Scalabrino Classifier - Comments Removed

	ChatGPT		Gemini		Llama		Claude		Deep
Metric	4o	4o mini	2.0 Pro	2.0 Flash	3.1 405B	3.1 8B	3.7 Sonnet	3.5 Haiku	**Seek** V3
Precision	69.44%	69.44%	69.44%	69.44%	69.44%	69.44%	69.44%	69.44%	69.44%
Recall	83.33%	83.33%	83.33%	83.33%	83.33%	83.33%	83.33%	83.33%	83.33%
F1-Score	75.76%	75.76%	75.76%	75.76%	75.76%	75.76%	75.76%	75.76%	75.76%

4.4 Analysis after using confusing names

In the third analysis, Sonar's rating remained A and no attributes changed, making this scenario invisible to Sonar's analysis. The new distribution of LLM scores showed a noticeable decline compared to previous versions. This indicates that the naming of variables and methods in the code impacts the perception of readability according to the LLM evaluations.

All models changed its variability behavior upwards, Gemini 2.0 Flash, Claude 3.7 Sonnet and Deep Seek V3 presented no variation. Below we list the variation on score attribution in this scenario for the other models, with their respective SD and CV:

Claude 3.5 Haiku - C_2 5.2 (5.67%);
Llama 3.1-405B - C_5 1.0 (2.38%);
Llama 3.1-8B - C_3 6.3 (8.11%) and C_7 9.7 (13.06%);
ChatGPT 4o-mini - C_1 4.2 (5.78%), C_6 3.2 (4.94%) and C_{11} 4.2 (5.78%);
ChatGPT 4o - C_1 2.6 (3.80%), C_3 and C_4 5.2 (6.54%);
Gemini 2.0 Pro - C_2 2.1 (2.24%), C_4 4.2 (6.30%), C_5 4.2 (19.17%), C_6 3.2 (10.20%) and C_9, C_{11} 1.6 (2.61%);

Table 7 presents the scores obtained by class and LLM for this analysis. Almost all evaluations showed a score reduction highlighted in red. ChatGPT 4o presented two reductions with moderate statistical significance, in yellow. ChatGPT 4o-mini Llama 3.1-8B and Claude 3.5 Haiku presented one non statistical significant reduction each one and Gemini 2.0 Pro presented two, all of them in gray color.

Table 7: Average Score by Class and LLM - Use of Confusing Names

	ChatGPT		Gemini		Llama		Claude		DeepSeek	SC
Class	4o	4o-mini	2.0-Pro	2.0-Flash	3.1-405B	3.1-8B	3.7-Sonnet	3.5-Haiku	V3	
C_1	-28.4%	-14.1%	-68.4%	-38.9%	-57.9%	-7.6%	-77.8%	-10.5%	-57.9	-0.69%
C_2	-10.5%	-10.5%	-1.1%	-5.6%	-15.8%	-3.1%	-31.6%	-4.2%	-10.5	-1.72%
C_3	-7.1%	-11.8%	-29.4%	-47.1%	-29.4%	-15.2%	-58.8%	-11.8%	-11.8	-2.34%
C_4	-7.1%	-9.6%	-7.5%	-13.3%	-28.1%	0.0%	-52.9%	-11.8%	-23.5	0.13%
C_5	-11.8%	-13.3%	-70.7%	-60.0%	-42.0%	-25.0%	-53.8%	-13.3%	-52.9	-13.12%
C_6	-23.5%	-14.7%	-56.0%	-46.7%	-66.7%	0.0%	-64.7%	-23.5%	-47.1	0.85%
C_7	-20.6%	-17.6%	-26.3%	-78.9%	-29.4%	-7.5%	-58.8%	-23.5%	-31.6	-1.56%
C_8	0.0%	-11.8%	0.0%	-13.3%	-25.0%	0.0%	-58.8%	0.0%	-23.5	-2.34%
C_9	0.0%	0.0%	-30.9%	-11.8%	-29.4%	-25.0%	-60.0%	-23.5%	-11.8	1.11%
C_{10}	0.0%	0.0%	-17.6%	-5.6%	0.0%	0.0%	-20.0%	-11.8%	-11.8	-0.50%
C_{11}	-23.5%	-2.7%	-16.0%	-46.7%	-33.3%	-25.0%	-60.0%	-23.5%	-23.5	-1.25%
C_{12}	-11.8%	-13.3%	-60.0%	-40.0%	-42.9%	-29.4%	-53.8%	-23.5%	-52.4	-10.06%

Gemini 2.0 Flash, Claude 3.7 Sonnet and Deep Seek V3 were the only ones reducing scores for every class. The Llama 3.1-8B model maintained the score for the largest number of classes, followed by ChatGPT 4o and ChatGPT 4o-mini.

Table 8 reveals the increased and inconsistent divergence between LLMs readability scores with a reference model. Unlike previous scenarios, the agreement rates vary dramatically between models. They range from a high recall of 91.67% for models like ChatGPT 4o, ChatGPT 4o-mini, Llama 3.1-8B, and Claude 3.5 Haiku, to an extremely low 16.67% for Claude 3.7 Sonnet.

Gemini 2.0 Flash and Claude 3.7 Sonnet reacted most strongly to the nonsensical names. Claude 3.7 Sonnet exemplifies this with an extremely low recall of 16.67%, correctly identifying only two readable classes and marking all others as unreadable. This shows a deep semantic analysis that diverges sharply from the SC's assessment, resulting in a very low F1-Score (28.21%).

Gemini 2.0 Pro and Deep Seek V3 were more conservative. Their high precision (91.67%) shows that when they classified code as readable, they were very likely to be correct according to the SC. However, their lower recall (66.67%) means they were stricter than the SCO and flagged more code as unreadable.

Similar to previous scenario SCO marked almost all classes as readable, but one as neutral.

Table 8: Agreement with Scalabrino Classifier - use of confusing names

	ChatGPT		Gemini		Llama		Claude		Deep
Metric	4o	4o mini	2.0 Pro	2.0 Flash	3.1 405B	3.1 8B	3.7 Sonnet	3.5 Haiku	**Seek** V3
Precision	84.03%	84.03%	91.67%	91.67%	91.67%	84.03%	91.67%	84.03%	91.67%
Recall	91.67%	91.67%	66.67%	41.67%	58.33%	91.67%	16.67%	91.67%	66.67%
F1-Score	87.68%	87.68%	77.19%	57.29%	71.30%	87.68%	28.21%	87.68%	77.19%

4.5 Analysis after code smell fix

In the fourth analysis, SonarQube indicated a strong reduction in Code Smells, along with changes in comments and lines of code. In Table 9 we marked the measures reductions in red and increments in green. Four classes suffer a score reduction by our reference classifier, two received a increase and other two (marked with *) had a slight increase which was truncated, since Scalabrino classifier uses 16 digits score.

Only three models presented variation. The Gemini 2.0 Pro model had its results significantly more severely affected by its variation. Below we list the variation on score attribution in this scenario, with their respective SD and CV:

Claude 3.7 Sonnet - C_5 3.2 (4.27%);

Table 9: SonarQube values after code cleaning

Class	Clean Code C	I	A	R	D	L	SC
C_1	-	-	-	-	62.9%	294	90.7
C_2	-	-	-	-	63.4%	526	91.1
C_3	-	-	-	-	32.1%	155	93.3*
C_4	-	-	-	-	21.0%	586	78.9*
C_5	-	-	1	-	9.8%	177	70.2
C_6	-	-	3	-	10.3%	825	76.5
C_7	-	-	-	-	25.5%	75	93.3
C_8	-	-	-	-	24.6%	589	72.3
C_9	-	-	1	-	17.2%	132	80.2
C_{10}	-	-	-	-	38.2%	48	89.1
C_{11}	-	-	5	-	6.9%	364	73.2
C_{12}	-	-	2	-	5.7%	276	67.1

Deep Seek V3 - C_6 1.6 (1.67%);
ChatGPT 4o-mini - C_6 4.8 (6.19%) and C_{11} 4.8 (5.89%);
Gemini 2.0 Pro - C_6 2.3 (3.14%), C_7 1.8 (1.95%) and C_8 1.8 (2.19%);
Llama 3.1-8B - C_5 6.3 (10.20%), C_7 1.6 (1.87%), C_8 2.4 (2.96%), C_{10} 3.0 (3.42%) and C_{12} 1.6 (1.96%);

In Table 10, we see that LLMs showed 16 instances of score increases, highlighted in green, being six with no statistical significance, in gray color and two with moderate significance in yellow. Also, we have 14 instances of score reductions in red color, with three of these reductions with no significance in gray. Claude 3.7 Sonnet was the only one to not present statistical noise.

Table 10: Average score by class and LLM - Clean Code

	ChatGPT		Gemini		Llama		Claude		DeepSeek	SC
Class	4o	4o-mini	2.0-Pro	2.0-Flash	3.1-405B	3.1-8B	3.7-Sonnet	3-Haiku	V3	
C_1	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	5.6%	0.0	0.0	0.0%
C_2	0.0%	0.0%	0.0%	0.0%	0.0%	-3.1%	0.0%	0.0	0.0	0.13%
C_3	0.0%	0.0%	0.0%	0.0%	0.0%	-7.6%	0.0%	0.0	0.0	0.03%
C_4	0.0%	2.4%	9.6%	0.0%	1.8%	0.0%	0.0%	0.0	0.0	0.02%
C_5	0.0%	0.0%	-6.7%	0.0%	0.0%	-22.5%	13.8%	0.0	0.0	8.97%
C_6	0.0%	4.0%	3.5%	0.0%	0.0%	33.3%	-11.8%	0.0	0.0	-1.11%
C_7	0.5%	0.0%	-4.7%	0.0%	0.0%	5.6%	0.0%	0.0	-0.5	0.0%
C_8	0.0%	0.0%	15.0%	0.0%	0.0%	35.8%	-11.8%	0.0	0.0	-3.38%
C_9	0.0%	0.0%	-2.9%	0.0%	-5.9%	6.2%	0.0%	0.0	0.0	0.0%
C_{10}	0.0%	-11.8%	0.0%	0.0%	0.0%	1.6%	0.0%	0.0	0.0	0.0%
C_{11}	0.0%	9.3%	-2.8%	0.0%	0.0%	-25.0%	0.0%	0.0	0.0	-1.10%
C_{12}	0.0%	0.0%	0.0%	0.0%	0.0%	-5.3%	15.4%	0.0	0.0	-2.03%

For this scenario, Table 11 shows a result equivalent to the original code. All LLMs agree with SC, marking all classes as readable.

Table 11: Agreement with Scalabrino Classifier - Clean Code

	ChatGPT		Gemini		Llama		Claude		Deep
Metric	4o	4o mini	2.0 Pro	2.0 Flash	3.1 405B	3.1 8B	3.7 Sonnet	3 Haiku	Seek V3
Precision	100%	100%	100%	100%	100%	100%	100%	100%	100%
Recall	100%	100%	100%	100%	100%	100%	100%	100%	100%
F1-Score	100%	100%	100%	100%	100%	100%	100%	100%	100%

Table 12 shows the results of thematic coding performed on LLMs reasoning for their scores. The table quantifies the frequency of themes mentioned by the LLMs for each of the four scenarios, based on the inductive thematic analysis methodology described in Section 3. The results reveal how the LLMs' qualitative assessments changed in response to each intervention:

Original Code (OC): This column establishes the baseline, showing high praise for "Good Code Structure" (1030 mentions) and "Good Readability" (1068 mentions). Mentions for "Good Documentation" (287) and "Good Variable/Method Names" (346) were also strong.

Comments Removed (I1): This intervention caused the expected sharp decline in comments about "Good Documentation" (from 287 to 32) and an increase in "Lack of Documentation/Comments" (from 17 to 64) and Unprofessional Comments (from 35 to 0). Consequently, mentions of "Good Readability" also decreased from 1068 to 933.

Confusing Names (I2): This scenario had the most dramatic impact on the LLMs' reasoning. Mentions of "Poor Variable/Method Names" skyrocketed from 4 to 890, while "Poor Readability" jumped from 105 in the original code to 561. This demonstrates that the LLMs' reasoning directly identified the semantic failure of the intervention, confirming their sensitivity to poor identifier choices. Also "Good Readability" plummeted from 1068 to 438, and "Good Code Structure" dropped from 1030 to 599.

Code Smells Fixed (I3): Fixing code smells led to a decrease in mentions of "Bad Code Structure" (from 125 to 116) and a significant increase in praise for "Good Error Handling" (from 65 to 92), suggesting these were the types of issues addressed. However, fixing smells did not uniformly improve the perception of readability, as mentions of "Poor Readability" slightly increased (from 105 to 133).

Table 12: Code Evaluation Across Scenarios

Theme	OC	I1	I2	I3
Adherence to Best Practices	267	226	98	263
Good Implementation/Logic	237	216	105	256
Good Use of Language Features	78	88	15	89
Consistent Formatting	182	178	53	208
Inconsistent Formatting	2	22	25	8
Good Code Structure	1030	983	599	1115
Bad Code Structure	125	155	74	116
Good Documentation	287	32	161	250
Lack of Documentation/Comments	17	64	31	22
Unprofessional Comments	35	0	42	40
Good Error Handling	65	132	20	92
Poor Error Handling	19	72	18	28
Good Readability	1068	933	438	1021
Poor Readability	105	99	561	133
Good Variable/Method Names	346	457	111	390
Poor Variable/Method Names	4	2	890	22

5 Discussion

With each intervention in the code, the LLMs presented different results. In this section, we will discuss the results presented.

In the original code scenario, the LLMs showed a level of correlation with SC, except for Gemini 2.0 Pro and ChatGPT 4o.

After removing comments ($OC \rightarrow I1$), all models showed at least one score reduction. The LLMs proved to be more lenient than the SC, correctly identifying most readable classes but at the cost of misclassifying several unreadable ones as described in Table 6. SonarQube showed a slight increase in Code Smells, as explained in step 4.3, and also indicated a significant drop in the percentage of comments and the total number of lines in the classes. SC showed a consistent and strong decrease of score for all classes, except for C_5 that received a strong increment, those new scores resulted in two classes falling as 'neutral' classification.

As described by Good and Poor Readability on Table 12, LLMs appear to consider that the absence of comments is not sufficient to drastically reduce readability, in most cases. There may be some

other aspects of code that interact with the presence (or absence) of comments, which could interfere in the LLMs analysis.

Changing the code to use confusing identifier names ($OC \rightarrow I2$) caused a significant fluctuation in scores across all models as well as an increase in the magnitude of its variability. Both Gemini models and Claude 3.7 Sonnet showed the greatest drop in scores, being Claude 3.7 Sonnet more consistent. All models showed some level of sensitivity to the unconventional names used in the experiment, significantly stronger comparing with reference model. But there were cases when a model did not reduce the score, as for C_{10}, the smallest class with 48 lines. As expected, SonarQube did not show any changes in its indicators after the identifier names were altered. All models were capable to criticize the semantic conflict between the structure of the presented algorithms and the adopted names, as mapped by thematic coding.

The SC showed some level of sensitivity in this scenario. That could be explained by the fact that this classifier performs a correlation between the terms in comments and terms in code, measuring the overlap between both (CIC measure). The use of confusing names reduced this correlation. This algorithm also takes into account the use of full formed words present in dictionary, as we have chosen to use this type of change, this measurement was not affected. Another metric used is called Textual Coherence, that is a measurement of vocabulary overlap between all possible pairs of blocks of code [11]. Since we have chosen to use only seasonings and vegetables names, we have created an artificial coherence, but with no semantic meaning to the code. These changes mislead SC but not the LLMs .

Cleaning the code ($OC \rightarrow I3$) resulted in all LLMs agreeing with SC, returning to the same results of original code state. The thematic coding demonstrated that LLMs reacted to changes reducing comments about "Bad Code Structure", incrementing comments on "Good Error Handling", but with a small increase of comments about "Poor Readability".

To evaluate LLMs variability we used 12 classes, 9 LLM and 4 scenarios, resulting in 432 combinations and 4,320 LLM executions. All LLM presented variation at least in one scenario. 47 (10.88%) combinations out of 432 presented variation, Gemini 2.0 Pro was responsible by 18 (38.30%) of all variations occurrence, being the only to present variation in all four scenarios. Conversely, Gemini 2.0 Flash was the only one to present no variation throughout all executions.

31 (65.96%) variations were equal to or lower than 5% and scenario I2 (confusing names) had 9 variations above 5% followed by I3 (code cleaning) with 3. Near half of variations did not compromise the statistical relevance of emitted scores. Therefore most of LLM score (94.02%) does not present statistical noise.

The reference model uses textual features to compose its readability score[11], but LLMs have shown the ability to evaluate semantic aspects of texts present in code, revealing deeper sensitivity to identifiers name changes, not present in the reference model. This was shown in scenario I2.

The same characteristics seem to affect the I1 scenario. The reference model has shown to be more sensitive to comments removal, mostly due to its metrics that correlate code with comments. Conversely, the LLMs seemed to be capable to extract the easiness of code to be read, even without comments. This fact raises a question: does LLM have used also the code semantics to understand it, dispensing documentation for algorithm comprehension?

6 Limitations and threats to validity

Shadish, Cook, and Campbell [31] enlist several types of threats to validity common to quasi-experiment design. The main categories are: *Construct Validity, Internal Validity, External Validity* and *Statistical Conclusion Validity*. In this section we detail all mapped threats to validity, limitations and future works derived from this study, grouped by the above listed categories.

Construct Validity Source code readability is a complex construct to be explained and characterized. Failing to do so, could lead us to confound it with other aspects of source code quality as well as incorrect inference about the the correlations observed from every operation (intervention) into the source code. To mitigate this threat we performed an extensive literature review on the main related studies and adopted a recognized state of the art model obtained in a state of use that required no implementation or intervention to work. Even so it is know that readability models have a limited capability to assess code readability the same way humans do [33, 34], being this a existent threat to validity.

The adopted strategy was also a mitigating factor for other two threats: The mono-operation bias caused by our strategy to perform a single stereotyped operation at a time (i.e removing comments, replacing identifiers names), which under represent the broad concept of source code readability. And the mono-method bias caused by the self produced source code interventions, which could lead to a bias of interventions performed based on what the research team understands to be a *"code readability intervention"*.

There is a threat to confounding the constructs with the levels of constructs. In this quai-experiment, we performed interventions that were limited to a very specific set of characteristics related to source code readability. We are not able to generalize it to the broad concept of readability. The findings we present can lead us to conclusions related solely to those manipulated characteristics. The control mechanisms used guarantee that the interventions achieved the goal to change the source code readability, for those aspects solely.

Internal Validity We tried to select algorithms with different levels of readability. Since it was an arbitrary selection, we are subjected to the selection bias. To remediate any undesirable effect from this threat, we executed a pretest to all selected classes using all selected LLMs, comparing the results with the control measurement tools.

We also recognize that we are subject to the internal validity threat described by Shadish, Cook, and Campbell [31] as the *"Instrumentation Threat"*. Since the LLMs were used through web API, we have no control over its internal state and runtime configuration. The providers could change any parameter, causing a difference in its behavior which could be confused as a difference caused by code interventions. To prevent that, we have executed all API calls to the same LLM into the same day, in a short time interval.

External Validity The source code used in this work is written in Java, as well as one of the control tools used (SC), which was able to perform readability analysis for Java Code. This is a known threat

to validity of our findings, as the interventions executed could have different outcomes for different programming languages. Despite the fact the Java is the most studied language in this subject, it is important to reproduce the study with different programming languages in the future.

We have chosen not to study the interactions of causal relationship over code interventions (i.e treatment variations); this is a known limitation of the collected results. The SC model, for example, measures the relationship between algorithm terms and documentation terminology. By performing the interventions in isolation we introduced a bias to the results, as detailed in section 5. As the goal of this work is to characterize how LLMs respond to such isolated changes, this approach was considered enough to answer the research questions. In the future a study could be performed to analyze how the interaction between different code interventions affects the perception of readability.

Our work uses the results of Scalabrino's Classifier (SCO) as ground truth, despite evidence that it can be unreliable under certain conditions [42], and that it does not fully capture code readability as perceived by humans [44, 42, 34]. This is a known threat to validity of this work and a limitation. The results only prove the comparability of LLMs with SCO, even though they produce results semantically distinct.

Statistical Conclusion Validity Since we selected a small set of algorithms to perform manual and controlled changes, we are subject to a threat of having low statistical power in our results as well as other statistical threats, such as inaccurate results. To ensure the results described are adequate, we performed complementary statistical tests, using non parametric tests. With these tests we were able to detect non statistically relevant results, and highlight them during results description. Still, the low amount of code used poses a threat to the generalizability of the results obtained.

An extraneous variance in the experimental setting is a threat caused by the known variance of LLMs answers to a prompt. To mitigate this threat we performed a 10 fold execution for every evaluation and executed statistical analysis for the results. This approach allowed us to draw the conclusions on when a LLM score have changes because of a code intervention and when its just random noise.

The unreliability of treatment implementation is a threat that could raise from the manual interventions done into the code. In order to mitigate it, we decided to execute very simple interventions, as described in section 3.5, just removing the comments, replacing identifiers name by a dictionary of words, and performing SonarQube's guided fixing for Code Smells. With this approach we have guaranteed a strict and reproducible method for manual code intervention.

7 Conclusion and future work

This work explored the sensitivity of LLM models to interventions related to source code readability. Our results demonstrated that all models exhibited some level of reaction to the changes. However, the results also raised some new questions.

(1) The results of this research are promising regarding the possibility of using LLMs to generate a metric that assesses code quality attributes related to readability. However, when comparing the results demonstrated by LLMs with the reference model, we observe differences in their scores.

The LLM analysis did not fully agree with the reference model, but behaved differently. It was influenced by the changes performed on source code; but this resulted in a score statistically different from the reference model in most cases. As future work, we plan to compare the LLMs results with human evaluation.

(2) With respect to the attributes that affect readability, we observed that LLMs show more sensitivity to the changes made to identifiers name (I2), when intentionally confusing names have been used. This behavior raised a hypothesis that the LLMs are relying on semantic meaning of words to evaluate code quality.

When comments were removed (I1), it was expected to see a reduction in readability scores. Although there was some reduction (20.83% of subjects), the majority of scores remained unchanged, with even a few improvements. This behavior could also be caused by the reliance on semantic meaning, but now extracted from the algorithm itself. There may be other code attributes that could be influencing LLMs to generate a score for code readability. This should be investigated in future work.

The clean code scenario (I3) demonstrated disappointing results, with more score reductions (9) than improvements (5), a lot of statistical noise (11) and mostly no change (71). Even for the control tool SC there was 4 improvements, 4 reductions and 4 classes with no change. We conclude that fixing code smells does not guarantee an improvement in code readability.

(3) In relation to the inherent variability of LLMs, our results show that while this variability affects the assessment of code quality, its occurrence rate is low. In the four scenarios evaluated, there were 9.37% (OC and I1), 12.50% (I3) and 14.58% (I2) of combinations with standard deviation different from zero. Most of them affected the statistical reliability of results, with exception of scenario I2. We recommend future evaluations of LLMs based on this study, to perform additional redundant tests, followed by statistical analysis, to check the validity of obtained results. A ten round execution demonstrated to be sufficient to provide data for a statistical validity check.

Overall, this study demonstrates that LLMs have the potential to be used as a complementary technique for evaluating quality aspects involving semantics, such as identifier names, comments content, and code documentation. Further research is needed to investigate the comparability of the metrics generated by LLMs with human developers' perceptions and to develop strategies to address the variability in the model's responses and their activation costs.

Artifact Availability

The source code of the programs used to execute the LLMs, as well as the source codes of the analyzed programs and the datasets from these analyses, are available in the GitHub repository referenced in this article [38].

REFERENCES

[1] Darrell R. Raymond. "Reading source code". In: *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*. CASCON '91. Toronto, Ontario, Canada: IBM Press, Oct. 1991, pp. 3–16. (Visited on 05/02/2025).

[2] Michael P. O'Brien, Jim Buckley, and Teresa M. Shaft. "Expectation based, inference-based, and bottom-up software comprehension". en. In: *Journal of Software Maintenance and Evolution: Research and Practice* 16.6 (Nov. 2004), pp. 427–447. ISSN: 1532-060X, 1532-0618. DOI: 10.1002/smr.307. URL: https://onlinelibrary.wiley.com/doi/10.1002/smr.307.

[3] B. W. Boehm, J. R. Brown, and M. Lipow. "Quantitative evaluation of software quality". In: *Proceedings of the 2nd International Conference on Software Engineering*. ICSE '76. event-place: San Francisco, California, USA. Washington, DC, USA: IEEE Computer Society Press, 1976, pp. 592–605.

[4] Simone Scalabrino et al. "Automatically Assessing Code Understandability". In: *IEEE Transactions on Software Engineering* 47.3 (Mar. 2019), pp. 595–613. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2019.2901468.

[5] Philippe Kruchten and Ipek Ozkaya. *Managing Technical Debt Reducing Friction in Software Development*. eng. OCLC: 1338840518. Sydney: Pearson Education, Limited, 2019. ISBN: 978-0-13-564596-3.

[6] Lan Cheng et al. "What improves developer productivity at google? code quality". en. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore Singapore: ACM, Nov. 2022, pp. 1302–1313. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3558940. URL: https://dl.acm.org/doi/10.1145/3540250.3558940.

[7] Terese Besker et al. "The influence of Technical Debt on software developer morale". en. In: *Journal of Systems and Software* 167 (Sept. 2020), p. 110586. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110586. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121220300674 (visited on 08/09/2024).

[8] Marvin Wyrich, Justus Bogner, and Stefan Wagner. "40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study". en. In: *ACM Computing Surveys* 56.4 (Apr. 2024), pp. 1–42. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3626522. URL: https://dl.acm.org/doi/10.1145/3626522.

[9] Raymond P.L. Buse and Westley R. Weimer. "A metric for software readability". en. In: *Proceedings of the 2008 international symposium on Software testing and analysis*. Seattle WA USA: ACM, July 2008, pp. 121–130. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390647. URL: https://dl.acm.org/doi/10.1145/1390630.1390647 (visited on 01/27/2025).

[10] Jonathan Dorn. "A general software readability model". In: *MCS Thesis* 5 (2012), pp. 11–14. URL: http://www.cs.virginia.edu/weimer/students/dorn-mcs-paper.pdf.

[11] Simone Scalabrino et al. "Improving code readability models with textual features". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. Austin, TX, USA: IEEE, May 2016, pp. 1–10. ISBN: 978-1-5090-1428-6. DOI: 10.1109/ICPC.2016.7503707. URL: http://ieeexplore.ieee.org/document/7503707/.

[12] Simone Scalabrino et al. "A comprehensive model for code readability". en. In: *Journal of Software: Evolution and Process* 30.6 (June 2018), e1958. DOI: 10.1002/smr.1958. URL: https://onlinelibrary.wiley.com/doi/10.1002/smr.1958.

[13] Qing Mi et al. "An Inception Architecture-Based Model for Improving Code Readability Classification". en. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. Christchurch New Zealand: ACM, June 2018, pp. 139–144. ISBN: 978-1-4503-6403-4. DOI: 10.1145/3210459.3210473. URL: https://dl.acm.org/doi/10.1145/3210459.3210473 (visited on 02/15/2025).

[14] Qing Mi et al. "Towards using visual, semantic and structural features to improve code readability classification". en. In: *Journal of Systems and Software* 193 (Nov. 2022), p. 111454. ISSN: 01641212. DOI: 10.1016/j.jss.2022.111454. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121222001467.

[15] Chao Hu et al. "How Effectively Do Code Language Models Understand Poor-Readability Code?" en. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Sacramento CA USA: ACM, Oct. 2024, pp. 795–806. ISBN: 979-8-4007-1248-7. DOI: 10.1145/3691620.3695072. URL: https://dl.acm.org/doi/10.1145/3691620.3695072 (visited on 05/31/2025).

[16] Igor Regis Da Silva Simões and Elaine Venson. "Evaluating Source Code Quality with Large Languagem Models: a comparative study". en. In: *Proceedings of the XXIII Brazilian Symposium on Software Quality*. Salvador Bahia Brazil: ACM, Nov. 2024, pp. 103–113. ISBN: 979-8-4007-1777-2. DOI: 10.1145/3701625.3701650. URL: https://dl.acm.org/doi/10.1145/3701625.3701650 (visited on 01/01/2025).

[17] Antonio Vitale et al. *Personalized Code Readability Assessment: Are We There Yet?* arXiv:2503.07870 [cs]. Mar. 2025. DOI: 10.48550/arXiv.2503.07870. URL: http://arxiv.org/abs/2503.07870 (visited on 03/23/2025).

[18] Andreas Bexell. "Software Source Code Readability: A Mapping Study". PhD thesis. Karlskrona, Sweden: Blekinge Institute of Technology, Aug. 2020. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1452612&dswid=-8633.

[19] Xinyi Hou et al. "Large Language Models for Software Engineering: A Systematic Literature Review". en. In: *ACM Transactions on Software Engineering and Methodology* 33.8 (Nov. 2024), pp. 1–79. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3695988. (Visited on 05/29/2025).

[20] Raymond P L Buse and Westley R Weimer. "Learning a Metric for Code Readability". In: *IEEE Transactions on Software Engineering* 36.4 (July 2010), pp. 546–558. DOI: 10.1109/TSE.2009.70. URL: http://ieeexplore.ieee.org/document/5332232/.

[21] Jianyu Zhao et al. *Understanding Programs by Exploiting (Fuzzing) Test Cases*. Version Number: 2. 2023. DOI: 10.48550/ARXIV.2305.13592. URL: https://arxiv.org/abs/2305.13592.

[22] Man-Fai Wong et al. "Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review". en. In: *Entropy* 25.6 (June 2023), p. 888. ISSN: 1099-4300. DOI: 10.3390/e25060888.

[23] Yue Wang et al. "CodeT5+: Open Code Large Language Models for Code Understanding and Generation". In: *Proceedings*

of the 2023 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Dec. 2023. DOI: 10.18653/v1/2023.emnlp-main.68. URL: https://aclanthology.org/2023.emnlp-main.68/.

[24] Dung Nguyen Manh et al. "The Vault: A Comprehensive Multilingual Dataset for Advancing Code Understanding and Generation". In: *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*. Singapore: ACL Association for Computational Linguistics, Dec. 2023. URL: https://aclanthology.org/2023.nlposs-1.25.pdf.

[25] Simone Scalabrino et al. "Automatically assessing code understandability: How far are we?" In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana, IL: IEEE, Oct. 2017, pp. 417–427. ISBN: 978-1-5386-2684-9. DOI: 10.1109/ASE.2017.8115654. URL: http://ieeexplore.ieee.org/document/8115654/ (visited on 08/08/2024).

[26] Asher Trockman et al. ""Automatically assessing code understandability" reanalyzed: combined metrics matter". en. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. Gothenburg Sweden: ACM, May 2018, pp. 314–318. DOI: 10.1145/3196398.3196441. URL: https://dl.acm.org/doi/10.1145/3196398.3196441.

[27] Aditya Kanade et al. "Learning and Evaluating Contextual Embedding of Source Code". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 5110–5121. URL: https://proceedings.mlr.press/v119/kanade20a.html.

[28] Da Shen et al. *Benchmarking Language Models for Code Syntax Understanding*. Version Number: 1. 2022. DOI: 10.48550/ARXIV.2210.14473. URL: https://arxiv.org/abs/2210.14473.

[29] Qing Mi et al. "Improving code readability classification using convolutional neural networks". en. In: *Information and Software Technology* 104 (Dec. 2018), pp. 60–71. ISSN: 09505849. DOI: 10.1016/j.infsof.2018.07.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584918301496.

[30] Qing Mi et al. "An Enhanced Data Augmentation Approach to Support Multi-Class Code Readability Classification". In: July 2022, pp. 48–53. DOI: 10.18293/SEKE2022-130. URL: http://ksiresearchorg.ipage.com/seke/seke22paper/paper130.pdf.

[31] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. eng. Nachdr. Belmont, CA: Wadsworth Cengage Learning, 2001. ISBN: 978-0-395-61556-0.

[32] Claes Wohlin et al. *Experimentation in Software Engineering*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-29043-5 978-3-642-29044-2. DOI: 10.1007/978-3-642-29044-2. URL: http://link.springer.com/10.1007/978-3-642-29044-2.

[33] Sarah Fakhoury et al. "Improving Source Code Readability: Theory and Practice". In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. Montreal, QC, Canada: IEEE, May 2019, pp. 2–12. ISBN: 978-1-7281-1519-1. DOI: 10.1109/ICPC.2019.00014. URL: https://ieeexplore.ieee.org/document/8813254/.

[34] Agnia Sergeyuk et al. "Reassessing Java Code Readability Models with a Human-Centered Approach". en. In: *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. Lisbon Portugal: ACM, Apr. 2024, pp. 225–235. ISBN: 979-8-4007-0586-1. DOI: 10.1145/3643916.3644435. URL: https://dl.acm.org/doi/10.1145/3643916.3644435 (visited on 05/26/2025).

[35] Yupeng Chang et al. "A Survey on Evaluation of Large Language Models". en. In: *ACM Transactions on Intelligent Systems and Technology* 15.3 (June 2024), pp. 1–45. DOI: 10.1145/3641289. URL: https://dl.acm.org/doi/10.1145/3641289.

[36] Jason Wei et al. *Emergent Abilities of Large Language Models*. arXiv:2206.07682 [cs]. Oct. 2022. URL: http://arxiv.org/abs/2206.07682.

[37] Jules White et al. "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT". In: Monticello, Illinois, USA, Oct. 2023. URL: https://www.dre.vanderbilt.edu/~schmidt/PDF/PLoP-patterns.pdf.

[38] Igor Regis. *LLMSonarQuarkusAnalysis*. https://github.com/igorregis/LLMSonarQuarkusAnalysis. 2024.

[39] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. The Addison-Wesley object technology series. Reading, MA: Addison-Wesley, 1999. ISBN: 978-0-201-48567-7.

[40] Naser Al Madi et al. "From Novice to Expert: Analysis of Token Level Effects in a Longitudinal Eye Tracking Study". In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. Madrid, Spain: IEEE, May 2021, pp. 172–183. DOI: 10.1109/ICPC52881.2021.00025. URL: https://ieeexplore.ieee.org/document/9462965/.

[41] Andrea Schankin et al. "Descriptive compound identifier names improve source code comprehension". en. In: *Proceedings of the 26th Conference on Program Comprehension*. Gothenburg Sweden: ACM, May 2018, pp. 31–40. ISBN: 978-1-4503-5714-2. DOI: 10.1145/3196321.3196332. URL: https://dl.acm.org/doi/10.1145/3196321.3196332.

[42] Valentina Piantadosi et al. "How does code readability change during software evolution?" en. In: *Empirical Software Engineering* 25.6 (Nov. 2020), pp. 5374–5412. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-020-09886-9. URL: https://link.springer.com/10.1007/s10664-020-09886-9.

[43] Greg Guest, Kathleen MacQueen, and Emily Namey. *Applied Thematic Analysis*. 2455 Teller Road, Thousand Oaks California 91320 United States: SAGE Publications, Inc., 2012. ISBN: 978-1-4129-7167-6 978-1-4833-8443-6. DOI: 10.4135/9781483384436.

[44] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. "Improving Code: The (Mis) Perception of Quality Metrics". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Madrid: IEEE, Sept. 2018, pp. 80–91. ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00017. URL: https://ieeexplore.ieee.org/document/8530019/.