Evaluating Source Code Quality with Large Language Models: a comparative study

Igor Regis da Silva Simões
Computer Science Department
Universidade de Brasília
Brasilia, Distrito Federal, Brazil
Diretoria de Tecnologia
Banco do Brasil
Brasilia, Distrito Federal, Brazil
igor@bb.com.br

Elaine Venson
Computer Science Department
Universidade de Brasilia
Brasilia, Distrito Federal, Brazil
elainevenson@unb.br

Abstract

Code quality is an attribute composed of various metrics, such as complexity, readability, testability, interoperability, reusability, and the use of good or bad practices, among others. Static code analysis tools aim to measure a set of attributes to assess code quality. However, some quality attributes can only be measured by humans in code review activities, readability being an example. Given their natural language text processing capability, we hypothesize that a Large Language Model (LLM) could evaluate the quality of code, including attributes currently not automatable. This paper aims to describe and analyze the results obtained using LLMs as a static analysis tool, evaluating the overall quality of code. We compared the LLM with the results obtained with the SonarQube software and its Maintainability metric for two Open Source Software (OSS) Java projects, one with Maintainability Rating A and the other B. A total of 1,641 classes were analyzed, comparing the results in two versions of models: GPT 3.5 Turbo and GPT 4o. We demonstrated that the GPT 3.5 Turbo LLM has the ability to evaluate code quality, showing a correlation with Sonar's metrics. However, there are specific aspects that differ in what the LLM measures compared to SonarQube. The GPT 4o version did not present the same results, diverging from the previous model and Sonar by assigning a high classification to codes that were assessed as lower quality. This study demonstrates the potential of LLMs in evaluating code quality. However, further research is necessary to investigate limitations such as LLM's cost, variability of outputs and explore quality characteristics not measured by traditional static analysis tools.

CCS Concepts

• **Software and its engineering → Software verification and validation**; **Maintaining software**.

SBQS 2024, Salvador, Brazil

ACM ISBN 979-8-4007-1777-2/24/11
https://doi.org/10.1145/3701625.3701650

Keywords

Code Quality, Code Readability, Static Analysis, Software Engineering, LLM, ChatGPT.

ACM Reference Format:
Igor Regis da Silva Simões and Elaine Venson. 2024. Evaluating Source Code Quality with Large Language Models: a comparative study. In *XXIII Brazilian Symposium on Software Quality (SBQS 2024), November 05–08, 2024, Salvador, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3701625.3701650

1 Introduction

Before initiating a modification in software, developer need to read its source code to identify the target sections of the intervention and devise an implementation strategy that maintains coherence with the pre-existing source code. The coding process of the intervention then begins, which should follow a set of best practices. Software developers share practices for writing quality and readable code, also known as *Clean Code* [6, 23].

The quality and readability of the source code are factors that impact developer's performance and productivity. A study [4] with 2,139 developers found that improvements in code quality resulted in an increase in self-declared productivity perception by developers, this being the factor with the highest correlation among the 39 evaluated factors. Code quality is a measurable aspect and monitored through static analysis tools, such as SonarQube[1] and code linter tools.

Static analysis tools successfully identify signs of lack of quality in source code. These signs manifest themselves by the occurrences of the so-called *Code Smell* [9]. This term refers to a class of best practice violations and indicates instances in the code that suggest the existence of a problem requiring the developer's attention. These occurrences can be deterministic, originating from code structures, such as the existence of duplicate code, and are easily identified by existing tools.

LLMs are being investigated as a tool to assess code quality, as well as their ability to understand code. As an example, the benchmark dataset *CodeXGLUE (General Language Understanding Evaluation benchmark for CODE)*[22] evaluates the proficiency of LLMs in various tasks and programming languages, such as "Clone detection", "Defect detection", "Code completion", "Code translation", "Code search", "Code repair", "Code summarization" among

[1] https://www.sonarsource.com/products/sonarqube/

others [2]. The characteristics of this benchmark allow analyzing the ability of a model to *understand* code, but there is no specific evaluation regarding a model's ability to *evaluate* the quality of a code.

This work aims to identify the level of ability of LLMs in assessing source code quality detectable by tools, as of those that perform static analysis, by answering the questions:

- How do the results of an LLM compare to the results generated by a static analysis tool?
- What are the benefits and limitations of using LLM for code quality assessment?

To answer these questions, we conducted a quantitative comparative study of the quality rule analysis results from the SonarQube tool with the results obtained by an analysis performed by two versions of the LLM ChatGPT. To obtain analysis results similar to those produced by SonarQube, we defined a prompt commanding the LLM to issue a *score* ranging from 0 to 100 for the source code being analyzed, considering aspects of readability and overall code quality. The comparison between the quality results produced by SonarQube and the LLM was carried out through statistical analyses, seeking to identify the correlation between the generated values. With these results in hand, we analyzed the discrepancies between the LLM and SonarQube, and we were able to identify the limitations of the LLM compared to SonarQube, as well as the differentials obtained in its use.

The results of this study show that an LLM is capable of capturing aspects related to the overall quality of code, in agreement with what was pointed out by the reference tool. We found that there are characteristics of code quality captured by both the LLM and SonarQube, as well as characteristics captured exclusively by either the LLM or Sonar. We also verified differences in the calculations used by the LLM and SonarQube to assess code quality, raising the divergences of evaluation. Additionally, we observed that the LLM provided a finer analysis compared to the A to E categorization used by Sonar. However, divergent results were noted in different versions of the LLM (GPT 3.5 vs 4o), indicating that the results of this study should not be generalized to all LLMs.

The remainder of this paper is organized into the following sections: Section 2 addresses the concepts of static code analysis and LLM treated in this study; in Section 3, we present an overview of recently published research where LLM are applied in Software Engineering activities; following this, Section 4 describes the methodology used and Section 5 details the results obtained; finally, Section 7 presents the final considerations, pointing to future work from this study.

2 Background

2.1 Static Code Analysis

Static code analysis has become a fundamental part of many software development approaches [32]. It is present as a strategy to improve code quality and consequently optimize the Code Review process [2, 27, 31]. It is part of steps of modern compilers such as GraalVM [37], being necessary for the generation of optimized native code. It is present in popular tools like Github, which stores hundreds of millions of source code repositories and receives more than 1,000 pushes per minute, from more than 65 million developers. Even with the challenge of meeting this volume, human behavior is pointed out as the main difficulty related to the successful adoption of this type of tool[5].

The gamification of static code analysis has been studied in order to engage developers, thus addressing the human problem related to its adoption [25]. In industry-conducted research[8, 28], it is suggested that success in engaging the developer would be related to bringing static analysis closer to the workflow, providing quick and short feedbacks. It was also identified that the number of defects pointed out by static analyses, which were accepted by developers, would be the success metric to be pursued.

The high level of false positives and false negatives are detractors of this success metric. Added to this is the inability to identify semantic aspects in identifier names, restricting itself to the use of dictionaries and grammatical correction. Examples of these aspects are: Does the code comment assist in understanding the algorithm? Does the name of an identifier (variable, method/function) have relevant meaning to the rest of the purpose of the code, being self-descriptive? Such questions could only be answered by a human reviewer. Limiting the scope of static code analysis tools [10].

Gunawardena et al. [10] identified that 4.6% of the occurrences pointed out in a code review, refer to the naming of identifiers; 16.4% refer to API/Javadoc documentation and 4.8% to comments. Totaling 25.8% of the points of 417 code reviews analyzed. They also identified that a large portion of these occurrences are not susceptible to automated identification by static analysis tools, requiring a developer's review.

2.2 Large Language Model

Large Language Models (LLMs) are Artificial Intelligence models, some with tens of billions of parameters, trained using a large volume of data. The success in their development and use is marked by the architecture called *Transformer* [34], which proposes an attention mechanism. The LLMs have demonstrated their superiority over previous models, enabling faster training and better results in benchmark tests. Conventional AI operates on regression and classification problems, analyzing data, classifying and identifying patterns. LLMs possess the capabilities of conventional AI, adding what is being called generative AI, which expands this scope of action to create content. Generative AI currently are able to generate text, image, sound and video, allowing LLMs to understand commands in text and generate responses in text [40]. The use of LLM in software engineering activities has increased with the emergence of new research and solutions in addition to its popularization [14].

3 Related work

There is a rich research scenario, with more than 70 different LLMs in use, of which 45 follow the architecture called *decoder-only*, just like its most popular example, ChatGPT [14]. Hou et al. [14] mapped studies related to the use of LLMs in Software Engineering, covering the period from 2017 to 2023. When analyzing the distribution of primary articles, they identified that the highest concentration is in the Development activity, with the task of generating code from a natural language prompt comprising the majority of the articles.

[2]Microsoft. CodeXGLUE. https://github.com/microsoft/CodeXGLUE

The second highest concentration is in Program Repair articles, which study the identification and repair of defects in code. Code completion occupies the third position, popularized by programming assistants like Github Copilot, focused on helping the developer create code. The fourth category is dedicated to the activity of explaining the code, passed as a parameter, in natural language. In the fifth and sixth positions are respectively: The activity of vulnerability detection, in which it is sought to use LLMs to detect security weaknesses; and the generation of tests, with emphasis on the largest number of research related to the generation of unit tests.

These tasks suggest the ability of LLM's to understand and analyze codes, but without a dedicated study to evaluate this capability. The task of static analysis is directly related to this capability and has only two articles identified in the study, presented below.

Hao et al. [12] present the results obtained in a prototype, using an LLM that must execute a pseudocode in order to perform static analysis. The developed prototype receives as a parameter, the pseudocode that is passed as a prompt to the LLM, along with information about the task to be performed. The third parameter is the source of the program to be analyzed. The article demonstrates the effectiveness of the proposed technique, comparing its results with the results obtained in the use of the LLM without the pseudocode-based technique. There was a 52.94% improvement in overall accuracy. There was also an improvement in token consumption, the use of the technique presented 46.06% of the tokens without the use of the technique.

Mohajer et al. [24] present the results obtained with a static code analysis tool endowed with capabilities of an LLM. It has the ability to identify defects through static analysis with LLM. It also filters false positives in an additional step with the use of LLM and finally performs the correction of the defects found, with modifications also made by LLM. The evaluation of the results was done using two common types of defects, *Null Deference* and *Resource Leak*. The defect detection capability was compared with that of the static analysis tool Infer[3]. The detection of *Null Deference* was superior to Infer, and the hit rate for *Resource Leak* was also higher. The accuracy in identifying false positives increased significantly for both cases. After analysis with LLM, the capability for *Null Deference* improved, and for *Resource Leak* there were no false positives. The automatic correction with LLM presented high correctness for *Null Deference* and *Resource Leak*, with almost all the generated code being syntactically correct.

Both articles demonstrate promising results with the use of LLM for static code analysis. However, the focus of both is on defect detection. The ability of LLMs in static analysis aimed at overall code quality, which is the objective of this study, was not evaluated.

The systematic mapping of Hou et al. [14] also identified seven articles classified as related to the Code Review task of the Maintenance activity. Among them, two articles deal with performing program editing based on Code Review comments [41] and evaluating the readability and explainability of programs generated by LLM's [20].

Five other studies, [11, 19, 21, 29, 33], are dedicated to automating Code Review. They use LLM to identify points of improvement in the code and generate review information.

Sghaier and Sahraoui [29] evaluated the effectiveness of LLM's, pre-trained and fine-tuned, in predicting reviews from the source code. Tufano et al. [33] uses a Text-To-Text Transfer Transformer (T5) model to perform reviews recommending improvements to the code and perform the proposed changes. Li et al. [19] use the T5 model from the work of Tufano et al. [33], fine-tuned for Java code, as part of a framework (AUGER) presented in the study, comparing it to the LSTM, CopyNet and CodeBERT models. Lu et al. [21] use a LLaMA model in the composition of their framework to automate the Code Review process, comparing it to seven other LLMs.

Four of these works, [19, 21, 29, 33], performed the pre-training and fine-tune of the models and compared the results to other models, used as baselines. Code Review datasets were also used to execute the experiments.

The fifth study, conducted by Guo et al. [11], is the only one that evaluates a pre-trained model without performing fine-tune. The chosen model is ChatGPT, in its GPT-3.5-turbo and GPT-4 versions. A reference model, specialized in code review, was used, using a code review dataset. To mitigate the risk that the GPT models have used the public datasets in their training, an additional dataset was used. Five prompt strategies were evaluated: (P1) simplest prompt, (P2) P1 + scenario description, (P3) P1 + detailed requirements, (P4) P1 + concise requirements and (P5) P4 + scenario description. Five temperature levels were also evaluated: 0, 0.5, 1.0, 1.5 and 2.0. The best results were obtained with the combination of temperature = 0 and prompt P2. The P2 prompt used the prompt pattern named by White et al. [36] as Persona. The performance of the ChatGPT models in performing code refinements based on review comments was evaluated, similar to the work of Zhang et al. [41]. Its ability to analyze the quality of the code that underwent review, which is the focus of this study, was not evaluated. The results obtained were promising, but still limited.

4 Methodology

This observational study compares code quality metrics generated from the use of LLMs with metrics generated by the static analysis tool SonarQube.

This section describes the procedures used for: (1) selection of projects and classes to be analyzed; (2) selection of metrics used for comparison; (3) definition of the prompt to extract the metrics from the LLM; and (4) comparison of results.

4.1 Selection of Projects and Classes

The SonarCloud site[4] provides access to public analyses for OSS projects. At the time of this research, the language with the highest number of analyzed projects available was Java (excluding HTML). Java is also the most studied language for source code metrics [26] and code comprehensibility [38]. Two projects with a large volume of source code lines were chosen, one project with a maintainability rating of A and another with rating B, according to Sonar. Projects with an A rating have a technical debt cost for remediation equal to or less than 5% of the total software cost. For remediation costs

[3]Meta. Infer Web Site URL: https://fbinfer.com

[4]SonarCloud https://sonarcloud.io/explore/projects

between 6% and 10%, the project receives a B rating. The same criterion applies to each Java class file.

The first filter of the tool was applied, for projects with more than 500 Kloc, returning 86 projects. A second criterion was applied, selecting only renowned OSS projects (with more than 10k stars), leaving the following projects: Apache Hadoop, Druid, Spark, Flink, ElasticSearch and Quarkus. A third condition was applied, selecting projects with analyses carried out in the 10 days prior to this work and no divergence of activity date of the repository on Github and the last analysis of Sonar, leaving only the Quarkus. No project with rating B was found among the 86. We expanded the filter to the second interval allowed by the tool (100 - 500 Kloc), returning 978 projects. Seven projects were found on SonarCloud with a B maintainability rating, of which five were excluded because they did not have an analysis carried out in 2024 or because they had a divergence of activity date of the repository on Github and the last analysis of Sonar, which would lead the LLM to analyze a different source from Sonar. And finally, one project was excluded for not having a source available on Github, leaving only one project with a B rating.

The selected projects were Quarkus[5], a Cloud Native, (Linux) Container First framework for writing Java applications, and Shattered Pixel Dungeon (SPDungeon)[6], an open-source traditional roguelike dungeon crawler with randomized levels and enemies, and hundreds of items to collect and use. It's based on the source code of Pixel Dungeon, by Watabou. Table 1 presents the main characteristics of the two projects.

Table 1: Characteristics of the selected projects

Characteristic	Quarkus	SPDungeon
Sonar Rating	A	B
Github stars	13.4K	4.5K
Releases	319	47
Lines of code	725 KLoc	134 KLoc
Total of classes	11.493	1.790
Analyzed classes	644	997

The analyses carried out on SonarQube for the SPDungeon project are from a Github fork, with the same sources as the original repository.

The analysis was conducted for the project classes with the highest possible number of lines of code within the LLM token limit, resulting in files with fewer than 800 lines of source code. Automated test classes and automatically generated classes were disregarded. To preserve the state of the sample used in this study, the source code of each class was copied to a project on Github [7].

4.2 Analyzed Metrics

SonarQube offers a range of code metrics resulting from its analysis. The following metrics were selected for comparison (in parentheses is indicated how they will be referenced in this article):

(1) Number of Code Smells of a class (Code Smells).
(2) Comment Lines Density (% Comments).
(3) Cognitive complexity (Cog. Complexity).
(4) Complexity (Complexity).
(5) Lines (Lines) representing the amount of code.
(6) Statement (Statement), number of statements of a class.

Those metrics are some of the most used in researches related to code source code metrics [26] also measured by SonarQube. The exceptions are Code Smells being a composition of many metrics and Cognitive Complexity as a new metrics introduced into the tool [7] as a complementary metric to Cyclomatic Complexity with a enfasis to code understandability [3].

4.3 LLM's Prompt Definition

The primary LLM used for this study was ChatGPT 3.5 Turbo. This version presents a token cost 60 times lower than ChatGPT 4 and 10 times lower than the latest version, ChatGPT 4o. Laskar et al. [17] identified that newer versions of ChatGPT do not guarantee better results. The aim of this study was not to compare different LLMs, therefore the version with the lower cost was primarily used. As previous research found that newer versions may present worse results, an evaluation was also carried out with the latest version of ChatGPT, the 4o.

The analysis carried out with the LLM requires the elaboration of a suitable prompt for this purpose, the prompt of this study was based on specific patterns for ChatGPT [36] and the lessons learned by Guo et al. [11]. The first consists of guiding the LLM to assume a persona capable of performing the analysis with a certain knowledge. For this purpose, we elaborated 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 code 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 and overall quality of the source code presented."*.

To allow the evaluation of the score assigned by the LLM, we requested that it present the 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."*. This explanation have an auditing purpose only, by having the LLM to elaborate over the reasons of the generated score. To facilitate the reading of the LLM's responses and even its processing via software, we complemented it with the approach of defining a response template that the LLM must follow *"Your answers MUST be presented ONLY in the following json format: "score":"NN%", reasoning:"your explanation about the score""*.

We used three patterns mapped by White et al. [36] (*Persona, Template and Reflection*) to elaborate the prompt that guides the LLM on how to respond to the task assigned to it. The elaborated passages composed the command of the LLM as System Prompt: *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 sourcode 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 and overall quality of the sourcode presented. Your answers MUST be presented ONLY in the following json format: {"score":"NN%", reasoning:"your explanation about the score"} - The "explanation"*

[5]Quarkus https://github.com/quarkusio/quarkus
[6]SPDungeon https://github.com/ismvru/shattered-pixel-dungeon
[7]SonarQube Cognitive Complexity White Paper https://www.sonarsource.com/docs/CognitiveComplexity.pdf

attribute must not surpass 450 characters and MUST NOT contain especial characters or new lines.

In addition to the System Prompt, the User Prompt needs to be developed, a section in which the command that the LLM is expected to execute is given, and the code to be analyzed is inserted: *Evaluate the following Java source code: SOURCE-CODE This is the end of the class file, the assistant should present your json answer:*

Other parameters: According to the OpenAI documentation[8], the *Temperature* parameter regulates the degree of consistence of the LLM with reference to the command given to it. The value is a range between 0 and 2, where 0 regulates the LLM to offer factual responses and 2 regulates it to a more creative mode in which the LLM can generate responses without factual bases. Thus, for the task of code analysis, it is expected that the responses presented have a factual basis strictly related to the presented code, setting this parameter to the value 0 *(zero)*. Guo et al. [11] identified that this value generates the best results when performing prompts that seek to analyze code.

The context window of an LLM is determined by the combined length of its input and output. To maximize the number of lines of code into the input and avoid the truncation of the output, we limited the LLM response to 120 tokens, leaving the remaining tokens for the input (prompt + source code). A LLM can't count characters but solely tokens, the 120 token limit used as output parameter guarantee the generated response to be near to 450 characters [30].

4.4 Comparative Analysis

The results between the values of the metrics collected via Sonar and via LLM for the two projects were compared using Spearman correlation algorithms. Attributes such as mean, mode, standard deviation, median, and percentage of outliers were also measured to allow a better understanding of the data distribution. A listing of the distribution of ratings assigned by Sonar and scores generated by the LLM was made. This listing was cross-referenced to find discrepancies between both approaches. The discrepancies were analyzed and discussed.

5 Results

In this section, we present the result of the code quality analysis for each of the two selected OSS projects. For each of them, we present the characteristics of the analyzed classes, the result of the analysis from Sonar, the result of the analysis from the LLMs, and finally, the comparison of the results.

Table 2: Rating SonarQube

Rating	Classes
A	615
B	21
C	6
D	2
E	0

Table 3: Score LLM

Score	Classes
90	176
85	421
80	30
75	10
70	4
60	1
50	2

[8]OpenAI Documentation - https://platform.openai.com/docs/guides/text-generation/how-should-i-set-the-temperature-parameter

5.1 Project 1 Analysis - Quarkus

5.1.1 Selection of excerpts for analysis. Classes selected from the core layer of the source code of the Java project called Quarkus will be used, which already has analyses performed on the public site of SonarQube [9].

5.1.2 Sonar Analysis. The core layer of Quarkus has 644 classes that fit the selection criteria for this work, totaling 55,839 lines of code and 1,626 Code Smells. The distribution of the maintainability rating of the sample, presented in Table 2, indicates that the vast majority of the sample classes have a high maintainability index, with 95.49% of the classes in rating A and 3.26% in rating B, according to the metrics applied by Sonar.

Table 4 presents the summary statistics for the metrics collected from Sonar. With the exception of the percentage of comments, there is a high number of outliers in all attributes. Analyzing the statistical data of these attributes, it is noticed that most of the sample classes have no Code Smells, low complexity (both metrics), few lines of code, comments, and statements.

Table 4: Sonar's metrics statistics

Attribute	Average	Std.Dev.	Median	Mode	Outliers
Code Smells	2.52	5.14	0	0	9.78%
% Comments	13.92	16.37	7.7	0	4.35%
Cog. Complexity	10.16	25.49	0	0	15.22%
Complexity	11.53	19.81	4	0	12.27%
Lines	86.70	116.64	42.5	18	10.71%
Statements	24.25	47.87	5	0	12.73%

5.1.3 LLM Analysis. The number of classes for each score assigned by the LLM is listed in Table 3. There are 421 occurrences of score 85, representing 65.37% of the total, and 176 occurrences with score 90, accounting for 27.32%. A strong concentration of evaluations around score 85 is observed, with 97.36% of the classes having a score of 80 or higher.

Unlike the attributes derived from Sonar, for the score values produced by the LLM, there is a low occurrence of outliers, only 2.64% (score less than or equal to 70).

5.1.4 Results comparison. The SonarQube analysis shows that the group of classes with rating A corresponds to 95.49% of the sample, while the analysis with LLM shows that the group of classes with a score of 85 or higher corresponds to 92.70%, or 97.36% for a score of 80 or higher. The Rating assigned by SonarQube is a categorical variable that generated a concentration with value A. Figure 1 presents the apparent correlation between the distribution of the Sonar rating and the score assigned by the LLM.

Table 5 presents the relationship between the scores assigned by the LLM and the rating assigned by Sonar at the class level. It can be observed that there are discrepancies between the LLM and SonarQube, as exemplified by the cells highlighted in yellow. Two classes with a B rating received a score of 50. Both classes are empty, without attributes, methods, or comments, as can be observed in Listings[10] 1 and 2. In Listing 3, we have the score and the justification provided by the LLM for the assigned grade. It is

[9]Quarkus https://sonarcloud.io/summary/overall?id=quarkusio_quarkus
[10]We removed the package and import declarations

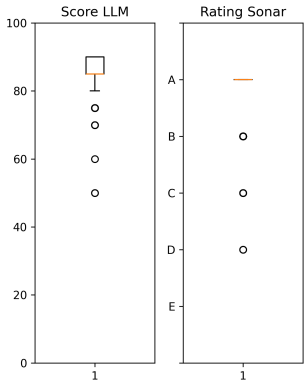

Figure 1: LLM scores vs Sonar rating

Table 5: Distribution of SonarQube vs LLM (Quarkus) reviews

LLM	Sonar Rating			
Score	A	B	C	D
50	0	2	0	0
60	1	0	0	0
70	4	0	0	0
75	9	1	0	0
80	27	1	2	0
85	404	13	2	2
90	170	4	2	0

notable that the absence of code and documentation determined the score assigned by the LLM.

Upon detailed analysis of the SonarQube evaluation, we found the existence of only one Code Smell of the *Maintenability* type for each class. It is a rule violation, considered of the *Minor* type by SonarQube. This rule warns about the use of empty classes in Java, considered a bad practice reducing code maintainability by generating comprehension confusion. The weight of the violation is given by SonarQube due to the estimated effort for the correction of the pointing, which for the case of this rule, is five minutes of work.

```
public class ConstPoolPredicate {
}
```

Listing 1: Classe ConstPoolPredicate (score 50)

```
public class Description {
}
```

Listing 2: Classe Description (score 50)

```
ConstPoolPredicate: {"score":"50","reasoning":"The code is very
    short and lacks any functionality or context. It is
    difficult to evaluate the quality of the code based on this
     class alone. However, the code follows standard Java
    naming conventions and is properly formatted."}
Description: {"score":"50","reasoning":"The code is very short
    and doesn't contain any logic. It's hard to evaluate the
    readability and quality of the code based on just one empty
     class."}
```

Listing 3: Samples of LLM's code reviews

We also analyzed the two classes with a D rating that received a high score (85). In Listings 4 and 5, we have the source of the classes. Unlike the *ConstPoolPredicate* and *Description* classes, these contain code and comments. The *SnapStartRecorder* class is short, but its content did not receive a negative evaluation from the LLM. In the justifications presented in Listing 6, we note a positive reference to the code structure, correct use of Java language capabilities, and a recommendation for documentation improvement, besides a comment about the good variable naming. The *QuarkusBindException* class received similar comments.

The *QuarkusBindException* class has only one violation, considered *Major*, with an estimated effort of four hours of work for correction. The *SnapStartRecorder* class has six violations of three SonarQube rules, totaling an estimated effort of one hour and 40 minutes. All violations compromise maintainability, two of the rules have a related CWE[11] [12], the other refers to the risk of problems in multi-threading scenarios[13]. None of the rules were pointed out by the LLM.

We conclude that the evaluation carried out by the LLM did not take into account the same aspects as SonarQube.

```
/**
 * An exception that is meant to stand in for {@link BindException} and
       provide information
 * about the target which caused the bind exception.
 */
public class QuarkusBindException extends BindException {
    private final List<Integer> ports;

    private static String createMessage(List<Integer> ports) {
        return "Port(s) already bound: " + ports.stream().map(i -> Integer.
    toString(i)).collect(Collectors.joining(", "));
    }

    private static void assertPortsNotEmpty(List<Integer> ports) {
        if (ports.isEmpty()) {
            throw new IllegalStateException("ports must not be empty");
        }
    }

    public QuarkusBindException(Integer... ports) {
        this(Arrays.asList(ports));
    }

    public QuarkusBindException(List<Integer> ports) {
        super(createMessage(ports));
        assertPortsNotEmpty(ports);
        this.ports = ports;
    }

    public QuarkusBindException(BindException e, Integer... ports) {
        this(e, Arrays.asList(ports));
    }

    public QuarkusBindException(BindException e, List<Integer> ports) {
        super(createMessage(ports) + ": " + e.getMessage());
        assertPortsNotEmpty(ports);
        this.ports = ports;
    }

    public List<Integer> getPorts() {
        return ports;
    }
}
```

Listing 4: Classe QuarkusBindException (score 85)

```
/**
 * Stores the optimizations to execute for SnapStart.
 */
@Recorder
public class SnapStartRecorder {
    public static boolean enabled = false;
    public static boolean fullWarmup = false;

    public void register(boolean fw) {
```

[11] Rule https://rules.sonarsource.com/java/RSPEC-1104/
[12] Rule https://rules.sonarsource.com/java/RSPEC-1444/
[13] Rule https://rules.sonarsource.com/java/RSPEC-2696/

```
10          enabled = true;
11          fullWarmup = fw;
12      }
13  }
```

Listing 5: Classe SnapStartRecorder (score 85)

```
1 QuarkusBindException: {"score":"85","reasoning":"The code is
      well-structured and follows Java conventions. The use of
      streams and lambdas is appropriate. However, the code could
       benefit from more comments and documentation to explain
      the purpose of the class and its methods."}
2 SnapStartRecorder: {"score":"85","reasoning":"The code is well-
      structured and easy to read. The variable names are clear
      and concise. However, the purpose of the class and its
      methods could be better documented."
```

Listing 6: Samples of LLM's code reviews

We used Spearman's correlation to test the correlation between the Sonar attributes and the score value of the LLM (Table 6). Spearman's correlation was used given the non-normal distribution of the data. Through this calculation, we noticed a moderate inverse correlation between the Code Smells, Cog Complexity, Complexity, Lines and Statements in relation to the LLM score. Higher values for those attributes correlate to lower levels of code quality and readability. This inverse correlation to LLM score confirms that the LLM score is positively correlated to source code quality. The % Comments showed a weak positive correlation and a level of statistical relevance that, although sufficient, is much lower than that presented in the other attributes.

Table 6: Spearman correlation for SonarQube metrics

Attribute	Correlation	p-value
Code Smells	-0.403	1.269e-26
% Comments	0.129	0.001
Cog. Complexity	-0.361	2.667e-21
Complexity	-0.330	6.785e-18
Lines	-0.390	7.880e-25
Statements	-0.356	9.286e-21

5.2 Project 2 Analysis - Shattered Pixel Dungeon

5.2.1 Selection of excerpts for analysis. Selected classes from the core layer of the project source code will be used, as was done for Project 1. The most recent analysis on the public SonarQube site [14] was carried out on 07/02/2024, which is the same date as the last activity of the repository on Github, thus ensuring symmetry between the sources used for the analysis with LLM and the Sonar data. The criterion used for the selection of classes and use of SonarQube attributes was the same as for project 1, the classes used in the analysis were also preserved.

5.2.2 Sonar Analysis. The core layer of SPDungeon contains 997 classes that fit the selection criteria for this work, totaling 148,701 lines of code and 4,432 Code Smells. Table 7 presents the distribution of the maintainability rating of the sample. Unlike the previous project, which had a concentration of 95.49% of classes with an A rating, this project presents a 67.2% concentration in this layer, in addition to presenting a higher occurrence of classes in the lower quality ratings.

[14] Shattered Pixel Dungeon https://sonarcloud.io/project/overview?id=ismvru_shattered-pixel-dungeon

Table 7: Rating SonarQube

Rating	Classes
A	670
B	109
C	86
D	103
E	29

Table 8: Score LLM

Score	Classes
90	1
85	107
80	301
75	467
70	117
60	3
50	1

Table 9: Sonar's metrics statistics

Attribute	Average	Std.Dev.	Median	Mode	Outliers
Code Smells	4.54	7.48	2	1	8.83%
% Comments	2.34	3.39	1.2	0	4.81%
Cog. Complexity	21.68	35.51	8	0	10.13%
Complexity	22.04	28.71	11	4	9.73%
Lines	149.14	131.72	101	51	9.83%
Statements	48.48	59.53	26	9	10.23%

Despite presenting a lower percentage of outliers than Project 1, there is still a high outliers occurrence, with the exception of the percentage of comments again, as per the data in Table 9. Analyzing the statistical data of these attributes, it is noticed that most of the classes in the sample have at least one Code Smell, a complexity per class double the value of Project 1 (both metrics), almost double the lines of code, even fewer comments, and double the statements.

5.2.3 LLM Analysis. The number of classes for each score assigned by the LLM is listed in Table 8. There are 108 occurrences of score 85, representing 10.73%, and one occurrence with score 90 representing 0.1% compared to 27.32% of the previous project. There is a strong concentration of evaluations around score 75 (46.84%), with 41.02% of the classes having a score of 80 or higher. The occurrence of outliers was even lower for Project 2, only 0.50%, but the outliers have scores of 90, 60, and 50, both sides of the scale.

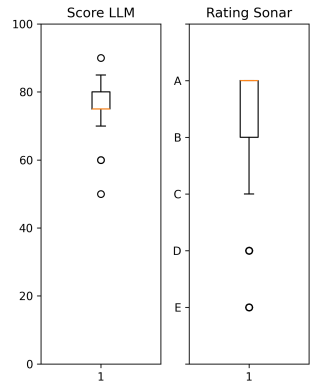

Figure 2: LLM score vs Sonar rating

5.2.4 Results comparison. The SonarQube analysis shows that the group of classes with an A rating corresponds to 67.2%, and another 29.88% are distributed among the B, C, and D ratings, while the analysis with LLM shows that the group of classes with a score of 80 or higher corresponds to 41.02%, and the scores between 70 and 75

correspond to 58.57%. For Project 2, we have a larger distribution of classes among various scores by the LLM. As in the Sonar analysis, in both cases there is a tendency to represent a lower code quality when compared to Project 1. In Figure 2, this approximation of both metrics towards the center is visually noticeable.

Table 10: Distribution of Sonar vs LLM (SP Dungeon) reviews

LLM	Sonar Rating				
Score	A	B	C	D	E
50	0	1	0	0	0
60	2	1	0	0	0
70	78	20	10	7	2
75	350	39	39	36	3
80	173	44	22	40	22
85	67	4	15	20	1
90	0	0	0	0	1

Analyzing the relationship of the LLM scores with each Sonar Rating, in Table 10, we note that even classes with a high rating (A and B) received low scores from the LLM. Two classes with a B rating received scores of 50 and 60 respectively, and two other classes with an A rating received a score of 60. No class with a C, D, or E rating received a score below 70, contrary to expectations. In addition, two classes with an E rating received scores of 85 and 90 respectively.

In Listing 7, we have the score and respective justifications provided by the LLM for the 4 classes with low scores. We note references to problems with code organization, indentation, and documentation, as well as some Code Smells. Some of the aspects pointed out are captured by SonarQube, such as the use of magic numbers or complex methods (cyclomatic complexity). Others are not identified by Sonar, such as methods that could receive better names, poorly formatted code, and appropriate documentation (SonarQube restricts itself to counting the percentage of documentation). The classes with these scores are: *SurfaceScene*, *Combo*, *ElementalStrike*, and *ShadowClone*, having between 430 and 580 lines of code. Because of its size, they will not be listed in the article, but can be consulted in the project related to this research, on GitHub [7].

```
1 SurfaceScene: {"score":"50","reasoning":"The code is poorly
      formatted and lacks comments, making it hard to read and
      understand. There are also some unused imports and
      variables, and some methods could be better named. However,
       the code seems to be functional and follows some good
      practices, such as using constants and separating concerns
      into different classes."}
2 Combo: {"score":"60","reasoning":"The code has good use of
      comments and follows a standard license. However, it lacks
      proper indentation and has some long methods and classes,
      which makes it hard to read and maintain. There are also
      some unused imports and some code smells, such as the use
      of public fields and the lack of encapsulation in some
      methods."}
3 ElementalStrike: {"score":"60","reasoning":"The code is well-
      organized and follows good naming conventions. However,
      there are some issues with readability due to the use of
      long method chains and nested if statements. Additionally,
      there are some areas where the code could be simplified and
       made more concise."}
4 ShadowClone: {"score":"60","reasoning":"The code has some good
      practices, such as the use of comments and proper
      indentation. However, there are some issues, such as the
      lack of proper documentation, the use of wildcard imports,
      and the presence of some long methods and classes.
      Additionally, there are some code smells, such as the use
      of magic numbers and the mixing of concerns in some methods
      ."}
```

Listing 7: Samples of LLM's code reviews

Table 11: Violations of SonarQube rules for classes on Listing 7

Classe	Code Smell	Bug
SurfaceScene	10	8
Combo	25	1
ElementalStrike	15	1
ShadowClone	10	0

The four classes presented various violations of SonarQube rules, some of the type Maintainability's Code Smell and Bugs related to Reliability, as listed in Table 11. The estimated time for correction of the violations was 27h30m, 6h, 4h36m, and 26h for the respective four classes. Some of the rules point to aspects cited by the LLM, such as long methods[15] and high cyclomatic complexity[16]. The large number of estimated hours to make corrections does not impact the rating of these classes due to their size.

The two classes with an E rating and scores of 85 and 90 have few lines of code, as per Listing 8 and 9. The *RedButton* class has only one violation with a repair effort of 4h30m. The *ParalyticDart* class has two violations, totaling 6h of effort. A common violation to the two classes refers to the rule *Inheritance tree of classes should not be too deep*[17], which points to the existence of a very deep inheritance hierarchy. This rule has a great weight in the calculation for repair cost, made by Sonar, and is invisible to the LLM, given that no information was passed beyond the source of the analyzed class. In Listing 10, we have the score and respective justifications of the LLM, pointing to little or no documentation in both classes and the magic number (line 11 of Listing 9) of *ParalyticDart*.

Again, it is concluded that despite overlap in some criteria, the LLM evaluates aspects that are not evaluated by SonarQube, and its score refers to the degree of readability while SonarQube refers to the cost of repair weighted by the size of the code.

```
1  public class RedButton extends StyledButton {
2
3    public RedButton( String label ) {
4      this(label, 9);
5    }
6
7    public RedButton( String label, int size ){
8      super( Chrome.Type.RED_BUTTON, label, size);
9    }
10
11 }
```

Listing 8: Classe RedButton (score 90)

```
1 public class ParalyticDart extends TippedDart {
2   {
3     image = ItemSpriteSheet.PARALYTIC_DART;
4   }
5
```

[15] https://rules.sonarsource.com/java/RSPEC-6541/
[16] https://rules.sonarsource.com/java/RSPEC-3776/
[17] https://rules.sonarsource.com/java/RSPEC-110/

```
6   @Override
7   public int proc( Char attacker, Char defender, int damage ) {
8     //when processing charged shot, only stun enemies
9     if (!processingChargedShot || attacker.alignment != defender.alignment) {
10      Buff.prolong(defender, Paralysis.class, 5f);
11    }
12    return super.proc( attacker, defender, damage );
13  }
14
15 }
```

Listing 9: Classe ParalyticDart (score 85)

```
1 RedButton: {"score":"90","reasoning":"The code is well-
      structured and follows good naming conventions. It also
      uses proper indentation and spacing. However, there is no
      documentation or comments to explain the purpose of the
      class and its methods."}
2 ParalyticDart: {"score":"85","reasoning":"The code follows good
      practices, such as proper indentation and comments. However
      , it lacks documentation and could benefit from more
      descriptive variable names. The code also has some hard-
      coded values that could be extracted to constants for
      better maintainability."}
```

Listing 10: Samples of LLM's code reviews

We used Spearman's correlation again to compare the correlation of the other attributes with the score value of the LLM (Table 12). The results obtained were similar to those of Project 1, with a significant deterioration in the identification of correlation for the % Comments attribute, which even began to show an inverse correlation, suggesting that comments have a slight influence on worsening the score.

Table 12: Spearman correlation for SonarQube metrics

Attribute	Correlation	p-value
Code Smells	-0.484	1.110e-59
% Comments	-0.149	2.163e-06
Cog. Complexity	-0.509	6.917e-67
Complexity	-0.536	2.783e-75
Lines	-0.567	6.663e-86
Statements	-0.557	1.836e-82

5.3 Analysis with second LLM

In the study by Laskar et al. [17], it was observed that newer versions of the ChatGPT model do not necessarily lead to better results. At the time of this research, the latest version of the model, called ChatGPT 4o, was released. Analyses were carried out with both projects using this version.

5.3.1 Quarkus Project. The data compared from GPT 3.5 and 4o (Table 13) show a greater tendency of GPT 4o to assign higher scores to the evaluated classes. As 95.49% (Table 2) of the classes have an A rating assigned by Sonar, GPT 3.5 presented 97.36% of the classes with a score of 80 or higher, while GPT 4o presented 99.22%.

In Table 14, the two classes that GPT 3.5 assigned a score of 50 were evaluated with a very low score by GPT 4o (score 10). The same classes considered of low quality by Sonar (rating D), which received a score of 85 by GPT 3.5, now received scores of 85 and 90 respectively. The cases of divergence were maintained, but a change in the score assigned by the LLM is noted, penalizing empty classes more severely.

Table 13: Scores GPT3.5 and GPT4o compared (Quarkus)

	Classes	
Score	GPT3.5	GPT4o
95	0	107
90	176	227
85	421	305
80	30	0
75	10	1
70	4	1
60	1	1
50	2	0
10	0	2

Table 14: Sonar vs GPT 4o (Quarkus)

LLM	Sonar Rating			
Score	A	B	C	D
10	0	2	0	0
60	1	0	0	0
70	1	0	0	0
75	1	0	0	0
85	285	15	4	1
90	220	4	2	1
95	107	0	0	0

Table 15: Scores GPT3.5 and GPT4o compared (SPDungeon)

	Classes	
Score	GPT3.5	GPT4o
90	1	1
85	107	990
80	301	0
75	467	5
70	117	0
65	0	1
60	3	0
50	1	0

Table 16: Sonar vs GPT 4o (SPDungeon)

LLM	Sonar Rating				
Score	A	B	C	D	E
65	0	1	0	0	0
75	4	0	1	0	0
85	665	108	85	103	29
90	1	0	0	0	0

5.3.2 SPDungeon Project. The data compared from GPT 3.5 and 4o (Table 15) again demonstrate a greater tendency of GPT 4o to assign higher scores to the evaluated classes, with 99.29% of the classes receiving a score of 85.

However, only 67.20% (Table 7) of the classes have an A rating assigned by Sonar, GPT 3.5 presented 41.02% of the classes with a score of 80 or higher, while GPT 4o presented a concentration of 99.39%. The results obtained with GPT 4o suggest that the readability and overall quality of the Shattered Pixel Dungeon project code is comparable or even superior to that of the Quarkus project. These results diverge from the evaluation made by SonarQube and the evaluation made by GPT 3.5.

When detailing the distribution of the LLM scores for each Sonar Rating, in Table 16, we note the existence of a low score assigned by the LLM for a class with a high rating. There is one occurrence for rating B with a score of 65. The same *AttackIndicator* class received a score of 70 from GPT 3.5, has 208 lines, and 5 Code Smells pointed out by SonarQube, totaling 4h47min of correction time. Again, some of the Code Smells were identified by the LLM and SonarQube, such as the FIXME comment, but others were identified only by SonarQube or the LLM. Regarding the classes with a low rating (D and E), the LLM assigned 100% of the scores at 85, GPT 3.5 had pointed out 3 classes with a score of 75 and 2 with a score of 70 (Table 10). The LLM was not able to differentiate the classes considered of lower quality according to Sonar and GPT 3.5 Turbo.

We conclude that GPT 4o presented results inferior to version 3.5 Turbo when executing the prompt evaluating the readability and overall quality of the code. This degradation confirms the findings of Laskar et al. [17].

6 Limitations and future work

This study conducted an evaluation on two open-source software (OSS) projects in the Java language. The number of evaluated projects may not be representative of the software industry. To

mitigate this risk, large-scale projects with a large volume of code for analysis were selected. One project with a SonarQube maintainability rating of A and another B were selected. No large-volume code projects with a C or lower rating were found. Future work could conduct an extensive study with a larger volume of OSS projects.

The Java language is widely adopted in the industry, making the findings of this research of broad interest. However, similar work needs to be carried out for other languages in the future.

The comparison base was Sonar, a tool widely adopted by the industry, however, there are other static analysis tools that provide maintainability-related metrics. Different results may be obtained when using other tools as a comparison base. This would occur if the criteria used to define maintainability diverge greatly between the tools. Future work could compare the results with LLM and other static analysis tools. The results can also be compared with other indices such as SQALE [18] used by Sonar, the TIOBE Quality Indicator (TQI) [16], SIG Maintainability[13], CISQ [1], and ISO/IEC 5055:202 [15].

A change in results was identified with different versions of the LLM model used (GPT 3.5 and 4o), confirming the finding made by Laskar et al. [17]. This factor limits the use of readability evaluations with LLM for benchmarks, being valid only to compare evaluations made by the same model and the same prompt. It is necessary to carry out an extensive comparative study between various LLMs and their results in code quality analysis, since other research [24] obtained better results in more recent versions of ChatGPT.

This study used the zero-shot technique. Different results may be obtained with the few-shot approach. Both techniques allow a model to perform tasks for which it was not explicitly trained [35]. Future research could investigate LLMs in conversational mode, to verify the accuracy and ability to analyze code that exceeds the context window as well as its evaluation of code from related classes, as proposed by Xia and Zhang [39] in their approach for APR.

This study compared the results obtained with the ChatGPT 3.5 turbo model. ChatGPT models are the most used in research, as identified by Hou et al. [14], with ChatGPT covered in 72 articles, ChatGPT 3.5 in 54, and ChatGPT 4 in 53. However, the literature review indicates an ecosystem of 45 variations of models and versions used in applied software engineering research since 2017. We understand the need to carry out research that compares various models and LLM architectures, in terms of their ability to assess readability and overall code quality.

LLMs present variability of responses for the same prompt. For the context of the current work, previous studies addressed and analyzed this variability, its impact, and strategies to deal with it[12]. We consider it important to evaluate the performance of various prompts in evaluating the application scenario proposed in this study.

For static analyses directed at specific problems, there are studies on prompt optimization[12]. This work evaluated the quality of code more broadly, yet limited to characteristics detectable by static analysis tools. It used researched practices of prompt engineering [36], but it is necessary to carry out studies for cost optimization in the use of LLM. LLM cost can be impeditive for its usage as a static analysis tool.

The results showed that SonarQube rate code quality based on the correcting cost of bad practices while hypothetically the LLM seems to rate the code based on the easy of reading. That hypotheses deserves a dedicated research. It is also needed a research dedicated to evaluate the execution time for a LLM to perform source code analysis and its feasibility of usage in industry constraints.

The LLM was capable of emit a score for the source code, limited to its context window. This impose a limit to the size of source code that can be analysed by a LLM. Future research is needed to investigate alternatives approaches to circumvent this limitation

7 Conclusion

This work explored the use of Large Language Models (LLMs) as a tool to evaluate code quality, comparing it to a static analysis tool. Two open-source software (OSS) projects were comparatively analyzed using SonarQube and two versions of the LLM GPT, covering a total of 1,641 classes.

We identified that ChatGPT 3.5 Turbo showed a correlation of its results with the maintainability score of SonarQube, demonstrating that this version of the LLM has the ability to evaluate the quality of a code, similarly to SonarQube. By analyzing the discrepancies, we found that the metric generated by the LLM corresponds to the readability and current quality of the code, being limited to the evaluation of the source code provided in the prompt. In contrast, the SonarQube maintainability rating refers to the estimated repair cost of the identified bad practices. SonarQube evaluates a broader set of rules, including relationships between classes (e.g., inheritance hierarchy depth). This cost is further weighed against the estimated cost of recreating the code from scratch. This is better detailed in section 4. Thus, classes with few lines of code have low tolerance to violations, receiving a low score from SonarQube if there is a violation that costs a few hours to correct, exceeding the creation cost of the analyzed class. The same class can receive a high score from the LLM, if the found violation is not detectable by the LLM or does not impact severely the readability or quality of the code.

The version of ChatGPT 4o presented inconsistent results when compared with its other evaluated version and with SonarQube result's. Leading to the conclusion that satisfactory results in one version of the LLM do not guarantee their occurrence in future versions of the model.

Limitations such as the cost per token and the variability of responses for the same prompt are still a challenge for the use of LLMs in substitution to static analysis tools to evaluate code quality. However, there is potential for their use in complement to these tools in order to observe quality characteristics not captured by them and enabling the creation of new quality indices.

This exploratory study evidences that LLMs can be used to evaluate the quality of source code, and can even expand the analysis currently carried out by static analysis tools by covering aspects that are currently not captured by them. New studies are necessary to investigate these different aspects of quality, as well as to propose mechanisms to deal with the cost and variability of outputs produced by LLMs.

References

[1] 2019. List of Weaknesses Included in the CISQ Automated Source Code Maintainability Measure. https://www.it-cisq.org/cisq-files/pdf/Maintainability-Weaknesses.pdf

[2] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 931–940. https://doi.org/10.1109/ICSE.2013.6606642

[3] G. Ann Campbell. 2018. Cognitive complexity: an overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*. ACM, Gothenburg Sweden, 57–58. https://doi.org/10.1145/3194164.3194186

[4] Lan Cheng, Emerson Murphy-Hill, Mark Canning, Ciera Jaspan, Collin Green, Andrea Knight, Nan Zhang, and Elizabeth Kammer. 2022. What improves developer productivity at google? code quality. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 1302–1313. https://doi.org/10.1145/3540250.3558940

[5] Timothy Clem and Patrick Thomson. 2021. Static Analysis at GitHub: An experience report. *Queue* 19, 4 (Aug. 2021), 42–67. https://doi.org/10.1145/3487019.3487022

[6] Maximiliano Contieri. 2023. *Clean Code Cookbook*. O'Reilly Media, Inc. https://www.amazon.com/-/dp/1098144724

[7] Igor Regis da Silva Simões. 2024. Github repository with preserved source code related to this paper. https://doi.org/10.5281/zenodo.13890179

[8] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[9] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley, Reading, MA.

[10] Sanuri Gunawardena, Ewan Tempero, and Kelly Blincoe. 2023. Concerns identified in code review: A fine-grained, faceted classification. *Information and Software Technology* 153 (Jan. 2023), 107054. https://doi.org/10.1016/j.infsof.2022.107054

[11] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. https://doi.org/10.1145/3597503.3623306

[12] Yu Hao, Weiteng Chen, Ziqiao Zhou, and Weidong Cui. 2023. E&V: Prompting Large Language Models to Perform Static Analysis by Pseudo-code Execution and Verification. http://arxiv.org/abs/2312.08477 arXiv:2312.08477 [cs].

[13] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A Practical Model for Measuring Maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. IEEE, Lisbon, Portugal, 30–39. https://doi.org/10.1109/QUATIC.2007.8

[14] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. http://arxiv.org/abs/2308.10620 arXiv:2308.10620 [cs].

[15] iso5055. 2021. ISO/IEC 5055:2021. https://www.iso.org/standard/80623.html

[16] Paul Jansen. 2023. The TIOBE Quality Indicator. https://www.tiobe.com/files/TIOBEQualityIndicator_v4_15.pdf

[17] Md Tahmid Rahman Laskar, M Saiful Bari, Mizanur Rahman, Md Amran Hossen Bhuiyan, Shafiq Joty, and Jimmy Xiangji Huang. 2023. A Systematic Study and Comprehensive Evaluation of ChatGPT on Benchmark Datasets. https://doi.org/10.48550/ARXIV.2305.18486 Version Number: 4.

[18] Jean-Louis Letouzey. 2016. The SQALE Method for Managing Technical Debt. http://sqale.org/wp-content/uploads/2016/08/SQALE-Method-EN-V1-1.pdf

[19] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. AUGER: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 1009–1021. https://doi.org/10.1145/3540250.3549099

[20] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2024. On the Reliability and Explainability of Language Models for Program Generation. *ACM Transactions on Software Engineering and Methodology* 33, 5 (June 2024), 1–26. https://doi.org/10.1145/3641540

[21] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Florence, Italy, 647–658. https://doi.org/10.1109/ISSRE59848.2023.00026

[22] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. https://openreview.net/forum?id=6lE4dQXaUcb

[23] Robert C. Martin (Ed.). 2009. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ.

[24] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2023. SkipAnalyzer: A Tool for Static Code Analysis with Large Language Models. http://arxiv.org/abs/2310.18532 arXiv:2310.18532 [cs].

[25] Lisa Nguyen Quang Do and Eric Bodden. 2018. Gamifying static analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista FL USA, 714–718. https://doi.org/10.1145/3236024.3264830

[26] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software* 128 (June 2017), 164–197. https://doi.org/10.1016/j.jss.2017.03.044

[27] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews?. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Montreal, QC, Canada, 161–170. https://doi.org/10.1109/SANER.2015.7081826

[28] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[29] Oussama Ben Sghaier and Houari Sahraoui. 2023. A Multi-Step Learning Approach to Assist Code Review. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Taipa, Macao, 450–460. https://doi.org/10.1109/SANER56733.2023.00049

[30] Andrew Shin and Kunitake Kaneko. 2024. Large Language Models Lack Understanding of Character Composition of Words. http://arxiv.org/abs/2405.11357 arXiv:2405.11357 [cs].

[31] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, 101–105. https://doi.org/10.1109/VLHCC.2017.8103456

[32] Patrick Thomson. 2021. Static Analysis: An Introduction: The fundamental challenge of software engineering is one of complexity. *Queue* 19, 4 (Aug. 2021), 29–41. https://doi.org/10.1145/3487019.3487021

[33] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2291–2302. https://doi.org/10.1145/3510003.3510621

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[35] Zhiqiang Wang, Yiran Pang, and Yanbin Lin. 2023. Large Language Models Are Zero-Shot Text Classifiers. http://arxiv.org/abs/2312.01044 arXiv:2312.01044 [cs].

[36] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. http://arxiv.org/abs/2302.11382 arXiv:2302.11382 [cs].

[37] Christian Wimmer. 2021. GraalVM native image: large-scale static analysis for Java (keynote). In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM, Chicago IL USA, 3–3. https://doi.org/10.1145/3486606.3488075

[38] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2024. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *Comput. Surveys* 56, 4 (April 2024), 1–42. https://doi.org/10.1145/3626522

[39] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. http://arxiv.org/abs/2301.13246 arXiv:2301.13246 [cs].

[40] Chaoning Zhang, Chenshuang Zhang, Sheng Zheng, Yu Qiao, Chenghao Li, Mengchun Zhang, Sumit Kumar Dam, Chu Myaet Thwal, Ye Lin Tun, Le Luang Huy, Donguk kim, Sung-Ho Bae, Lik-Hang Lee, Yang Yang, Heng Tao Shen, In So Kweon, and Choong Seon Hong. 2023. A Complete Survey on Generative AI (AIGC): Is ChatGPT from GPT-4 to GPT-5 All You Need? http://arxiv.org/abs/2303.11717 arXiv:2303.11717 [cs].

[41] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for Source Code and Natural Language Editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–12. https://doi.org/10.1145/3551349.3556955