Bad Smell Detection using Google Gemini

Larisse Amorim, Ivandeclei Mendes da Costa, Leticia Alves, and Eduardo Figueiredo

{larisseamorim, ivandeclei, leticiasma}@ufmg.br, figueiredo@dcc.ufmg.br
Software Engineering Laboratory (LabSoft), Federal University of Minas Gerais (UFMG), Brazil

***Abstract*—The detection of code smells is a critical task in software engineering, as it helps identify design issues that can compromise code quality and maintainability. With the advancement of large-scale language models (LLMs), such as Google Gemini, there is an opportunity to automate this detection more efficiently. This paper investigates the effectiveness of Gemini in identifying code smells in Java projects. We used a dataset (MLCQ) containing four types of code smells (Blob, Data Class, Feature Envy, and Long Method), classified into three severity levels. We then applied two types of prompts: a generic and a detailed one. To evaluate Gemini, we proposed three research questions related to its effectiveness with the different types of prompts used. Our results show that Gemini is more likely to provide correct results when using a detailed prompt compared to a generic one. Additionally, Gemini identified some code smells not highlighted by ChatGPT, suggesting their different detection capabilities. However, the accuracy of these additional detections still needs to be validated. We conclude that Google Gemini is a promising tool for code smell detection, but further studies are needed to understand its accuracy and the influence of the type of prompt used. This work paves the way for future research on the application of LLMs in software quality improvement.**

***Index Terms*—Code Smells, ChatGPT, Google Gemini**

I. Introduction

In recent years, Artificial Intelligence (AI) has seen remarkable growth, especially in the areas of Natural Language Processing (NLP), Machine Learning, and Computer Vision. This growth has led to major changes in several areas, including software engineering [26]. Looking at the AI recent advances, Large Language Models (LLMs) are gaining great visibility, as they have become a great ally in several areas of Software Engineering, such as software requirements [4], [5], automated test generation [5], and software vulnerability detection [7].

Inspired by a previous study [2], our goal is to explore the use of LLMs to detect code smells [1]. In this work, we evaluate the performance of the Google Gemini tool version 1.5 Pro for detecting code smells in JAVA project code. To perform this exploratory evaluation, we used an existing code smells dataset, called MLCQ [8], which contains 14,739 instances of code smells in GitHub repositories with four types: *Blob* which is the problem when a class becomes excessively large and complex, taking on multiple responsibilities that should be handled in separate classes; *Data Class* which is when a class mainly stores data without implementing significant functionality; *Feature Envy* which is when a method of one class excessively accesses the data or methods of another class and *Long Method*, which is when a method or function is excessively long and complex. Our aim is to answer the following questions:

- **RQ1:** Can Google Gemini identify code smells using a generic prompt?
- **RQ2:** Can Google Gemini identify code smells using a detailed prompt?
- **RQ3:** How does the effectiveness of Google Gemini differ between prompts?

Based on the responses we found, we analyze the efficiency of Google Gemini in detecting bad smells (inspired by a previous study on ChatGPT [2]). Our studies showed that Google Gemini was able to identify code smells in both prompts. When using the generic prompt, Gemini identified other types of smells that were not listed in the dataset, such as Duplicate Code, Lazy Class and Long Parameter List.

II. Background

This section introduces key notion of concepts that are being used in this work, namely Large Language Models (LLMs), code smells and the MLCQ dataset.

A. Large Language Models (LLMs)

Large Language Models (LLMs) are neural networks with the transformer architecture [3], pre-trained on massive textual content corpora and specifically tailored for text completion. Given textual inputs, for example, prompts, they generate corresponding text outputs in a probabilistic manner.

B. Code Smells

Code smells are particular bad patterns in source code which violate important principles of software design and implementation issues [1]. Particularly, code smells indicate when and what refactoring can be applied [6], [22]. This paper investigates four types of code smells discussed below.

- **Blob:** Also called a “God Class”, it is a class that is large and serves many different responsibilities, which are actively used throughout the code base.
- **Data Class:** it defines a class that is just a container for data, without any functionality.
- **Feature Envy:** this is a high severity smell [23] that occurs when a method accesses the data of another class more then its own data, relying more on members of other classes than its own.
- **Long Method:** this smell occurs when a method or a function is very long. This impacts the understandability and testability of the code [28].

C. MLCQ Code Smell Samples

The Machine Learning Code Quality Corpus (MLCQC) [25] is a dataset designed to address limitations found in previous research on code smells in the context of machine learning. Unlike other datasets, it was built in collaboration with professional software developers, ensuring greater relevance and quality. In addition, it includes data on the professional experience of the developers who participated in the review, allowing for deeper analysis of the perception of code smells in the industry [25].

MLCQ originally contains 14,739 code samples extracted from contemporary open source Java projects, with detailed information about four code smell type (Blob, Data Class, Feature Envy, and Long Method), classificated into four-level severity scales (None, Minor, Major, and Critical). Table I presents the detailed dataset instances. We discarded 11,448 instances classified as None in the original dataset; for this research, we analyzed the remaining 3,291 instances.

TABLE I
DATASET CONTENT

Severity	Blob	Data Class	Feature Envy	Long Method	Total
Minor	535	510	288	454	1,787
Major	312	401	142	274	1,129
Critical	127	146	24	78	1375
Total	974	1,057	454	806	3,291

III. Study Design

Figure 1 represents the data that we analyzed the code smells from MLCQ Code Smell Samples [8] (see II-C) and classified them into four severity levels: None, Minor, Major, and Critical. For the analysis, we discarded records classified as None, considering only the Minor, Major, and Critical levels. The dataset showed the following distribution: 375 records at the Critical level, 1,129 at the Major level, and 1,787 at the Minor level.

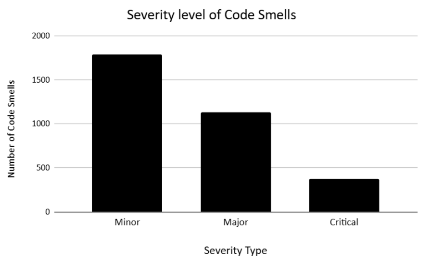

Fig. 1. Severity level of Code Smells found

A. Choosing Gemini

Figure 2 presents an overview of the study structure and the data processing flow. To analyze the dataset, we chose Gemini and use the prompts presented in the baseline study [2]. However, since our goal is to perform a replication, we decided to follow the same steps when modifying the LLM.

The reason for choosing Gemini is because it has a significant advantage in delivering factual information by its seamless integration with Google Search [9]. This integration allows the model to take advantage of Google's extensive knowledge base, resulting in more accurate and informative answers [9]. In addition, Gemini can effectively cite its sources, increasing the transparency and credibility of the information provided [9]. Another feature of Gemini is its advanced ability to understand complex instructions and maintain context in lengthy conversations. However, there are still small differences in the treatment of linguistic issues, which can limit its performance in some contexts [9]. The model also stands out for its Natural Language Processing (NLP) capabilities. Based on the Transformer architecture, introduced in 2017, Gemini relies on the attention mechanism to improve contextual awareness and language comprehension [9]. With its Retrieval-Augmented Generation (RAG) functionality, it combines information retrieval and text generation, offering fact-based outputs in a highly efficient manner [9]. In addition, Gemini has been trained on a diverse set of data and benefits from Google's scalable infrastructure [9]. This makes it versatile in a variety of tasks, including generating code and factual answers [9].

Despite its advantages, Gemini faces challenges related to biases inherent in the training data [9]. Although Google prioritizes bias mitigation and follows the "do no harm" principle, these problems still persist, reflecting the human biases in the real world data [9]. Another point of attention is the risk of the model producing biased or offensive content, even with efforts to ensure safety and to avoid harm [9]. This highlights the limitations of language models, even the most advanced ones like Gemini. While Gemini excels at producing accurate responses, especially when using the RAG functionality, it can occasionally generate problematic content due to the nature of its knowledge base [9].

B. Data Processing

We developed a .NET script to process the dataset in an automated manner, ensuring efficiency in the collection and analysis of information (Step 2 in Figure 2). The process begins by reading the dataset, where each line contains information about a specific source code associated with a code smell. For each entry, the script extracts the address of the repository on GitHub and accesses the corresponding source code. It then identifies and copies the code, where it contains the code smell. With this data in hand, the script combines the code with the prompt of the base study, creating standardized entries for analysis. After this step, the system sends the data to Gemini, which performs the analysis and generates a response based on the criteria established in the prompts. Finally, the script stores all responses in a CSV file, facilitating the organization and subsequent analysis of the results obtained.

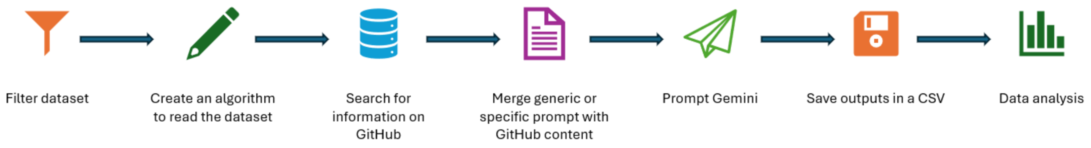
Filter dataset
Create an algorithm to read the dataset
Search for information on GitHub
Merge generic or specific prompt with GitHub content
Prompt Gemini
Save outputs in a CSV
Data analysis

Fig. 2. Steps ou our study

C. Used Prompts and Data Analysis

Table II shows the two types of prompts used in this study [2]: generic and detailed. The main difference between them is that, in the generic prompt, the LLM is instructed to identify code smells in the code provided, without specifying which ones should be found. In the detailed prompt, the code smells to be identified are clearly defined.

TABLE II
USED PROMPTS DESCRIPTION

Prompt Type	Description
Generic Prompt	I need to check if the Java code below contains code smells (aka bad smells). Could you please identify which smells occur in the following code? However, do not describe the smells, just list them. Please start your answer with “YES I found bad smells” when you find any bad smell. Otherwise, start your answer with “NO, I did not find any bad smell”. When you start to list the detected bad smells, always put in your answer “the bad smells are:” amongst the text of your answer and always separate it in this format: 1. Long method, 2. Feature envy [Java source code with the smell]
Detailed Prompt	The list below presents common code smells (aka bad smells). I need to check if the Java code provided at the end of the input contains at least one of them. • Blob • Data Class • Feature Envy • Long Method Could you please identify which smells occur in the following code? However, do not describe the smells, just list them. Please start your answer with “YES I found bad smells” when you find any bad smell. Otherwise, start your answer with “NO, I did not find any bad smell”. When you start to list the detected bad smells, always put in your answer “the bad smells are:” amongst the text of your answer and always separate it in this format: 1. Long method, 2. Feature envy ... [Java source code with the smell]

To analyze the data that was saved in the CSV file, considering the number of records that needed to be analyzed, we developed a .Net script that summed the number of code smells returned in each prompt. In the case of the generic prompt, Gemini often returned a variety of code smells and sometimes changed the output format. To deal with these inconsistencies, we configured the script to remove parts that were irrelevant to the analysis, such as explanations related to the meanings of the code smells found by Gemini.

IV. Analysis and Discussion

In this section, we present the analysis and discussion of the results found after executing the steps described in Section III. Section IV-A shows the efficiency of the script used for each prompt described in Table II. In Sections IV-B, IV-C, IV-D), we discuss each proposed research question.

A. Generic Prompt vs. Detailed Prompt

After running the script to read the dataset (see section III-B), we found that many repositories were unavailable. In addition, there was variation in the results related to processing errors, which may have been caused by various factors, such as failures in the GitHub service, the script being unable to open the page, or the unavailability of the Gemini service.

Figure 3 shows the difference between the prompts used. In the case of the generic prompt, Gemini indicated that, of the codes submitted, 111 had no code smells and recorded 428 processing errors. For the detailed prompt, Gemini reported that 262 code snippets did not have the defined code smells, while 381 errors occurred during script processing.

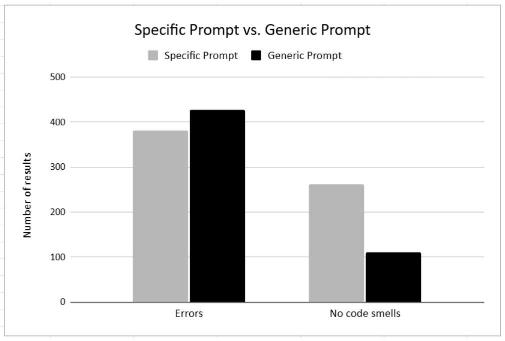

Fig. 3. Data processing

B. RQ1: Can Google Gemini identify code smells using a generic prompt?

After analyzing the data, we observed that Gemini, when using a generic prompt, was able to detect not only the code smells already catalogued (i.e., Data Class, Blob, Feature Envy, and Long Method), but also additional types of code smells such as Long Parameter List and Shotgun Surgery (see Table III). This suggests that the generic prompt enabled a broader and possibly deeper analysis of each code segment, often revealing multiple smells in a single code sample [6].

The possible discrepancy between Table III and Table III can be partially explained by this behavior. While Table III reflects all detections using the generic prompt, Table IV presents results filtered by specific prompts, which may have missed additional smells or were less sensitive to overlaps. Another possibility is that some false positives occurred in the generic prompt detections, although our analysis suggests that most additional smells were valid.

At this stage, it is not yet possible to determine with precision why certain code segments labeled with a specific smell were also flagged with others. Further investigation is needed to clarify whether these are genuine overlaps or limitations in the classification process. Nevertheless, the findings indicate that Gemini is capable of identifying a wide range of code smells, even when prompted generically.

TABLE III
NUMBER OF CODE SMELLS FOUND FOR THE GENERIC PROMPT

Code Smell	Total
Long Method	2496
Large Class	2117
God Class	1497
Data Class	1354
Long Parameter List	1179
Shotgun Surgery	837
Feature Envy	735
Primitive Obsession	524
Switch Statements	500
Duplicate Code	390
Refused Bequest	262
Lazy Class	119
Magic Number	68
Duplicated Code	55
Nested Blocks	37
Complex Method	34
Conditional Complexity	32
Nested Conditional	31
Nested Block Depth	24
Lazy Initialization	19
Message Chain	19
Middle Man	16
Inappropriate Intimacy	13
Nested Class	13

C. RQ2: Can Google Gemini identify code smells using a detailed prompt?

Table IV presents the number of bad smells detected in the detailed prompt by using Google Gemini. It shows that Gemini identified 188 classes containing the Blob smell, 415 classes containing the Data Class smell, 64 classes containing the Feature Envy smell and 2482 classes containing the Long Method smell. However, out of the 3291 classes contained in the dataset, Gemini was able to evaluate 3149 classes, corresponding to 95.68%. The reasons for not detecting code smells in the remaining about 5% of the data are due to one of two factors: 1) Gemini did not find any smells in the analyzed classes or 2) The script used to automate the process presented an error during execution and, thus, it skipped some classes.

D. RQ3: How does the effectiveness of Google Gemini differ between prompts?

When a generic prompt was used, in addition to the code smells listed in the dataset, Gemini was able to find other types of smells, as showed on Table III. However, there is a need to analyze the data to validate if the additional smells found are, in fact, smells present in the code.

TABLE IV
BAD SMELLS DETECTED IN DETAILED PROMPT

Smell	Total	(%)
Blob	188	5.71
Data Class	415	12.61
Feature Envy	64	1.94
Long Method	2482	75.42
Total	3149	95.68

Using a detailed prompt, Gemini correctly identified all the smells specified in the prompt based on the smells present in each instance in the database. Therefore, we recommend using Gemini with a detailed prompt where you can list the code smells of interest to check for their presence, since Gemini has shown satisfactory results for the instances it has analyzed. If you do not want to list the code smells of interest, you can use Gemini to list the code smells it identifies as being present in your code, but you will need prior knowledge and analysis to ensure that the answer provided by Gemini is correct.

V. THREATS TO VALIDITY

It is essential to recognize the limitations and potential threads of the study when interpreting the results. In this study, we found four validity topics that should be analyzed:

Internal Validity: During data analysis, processing errors were found that may have affected the results. The root cause of these errors could be failures in the GitHub service, problems accessing pages or unavailability of the Gemini service. The lack of control over these factors may have introduced biases into the data analyses.

Construction Quality: The dataset used contains code smells classified into three severity levels. However, the definition and classification of code smells can be subjective and vary between different developers. This may have influenced the accuracy of Google Gemini's detection of code smells.

External Validity: The study used a dataset with a limited number of code smells and focused on the Java programming language. The evaluation of Google Gemini on a larger and more diverse dataset is necessary to confirm the generalizability of the results. In addition, two prompt strategies were investigated, varying according to the level of detail. Other prompts could be evaluated in future work.

Conclusion Validity: An important question here is whether or not all (or most) of the smells found by Gemini are actually present in the code. If so, then one conclusion is that the generic prompt is more effective than the detailed one. If not, then the generic prompt actually opens the door for Gemini to find problems that do not actually exist.

VI. RELATED WORK

Large Language Models (LLMs) are Deep Learning-based systems designed to perform tasks related to Natural Language

Processing (NLP) [11] [12]. In the field of Software Engineering, researchers have explored the potential of LLMs to assist in various activities, such as automatic code completion [24], test case generation [13] [14], and code translation [12]. For instance, Liu et al. [11] proposed a pre-trained language model optimized for code completion, which outperformed state-of-the-art tools in Java and TypeScript programs. Similarly, Siddiq et al. [14] explored the use of LLMs for generating unit tests, achieving higher compilation success rates through heuristic-based prompts. While these studies demonstrate the versatility of LLMs in software engineering tasks, our work focuses specifically on their application in detecting code smells, an area that has received less attention in the literature. Unlike previous studies, we evaluate the effectiveness of Google Gemini in identifying code smells, exploring the impact of different prompt strategies.

This research differs from previous studies by using the same prompt applied in another study, with the aim of analyzing the behavior of Gemini. The preliminary analysis demonstrated that the model is capable of identifying code smells, which opens space for future comparisons with other LLMs. Furthermore, the results obtained allow us to investigate, strategies for the formulation of reusable prompts in different LLMs.

Pre-trained models have been widely used for code-related tasks, such as code review automation and program repair. Tufano et al. [17] evaluated the effectiveness of T5-based models in automating code reviews, demonstrating superior performance over traditional Deep Learning models. Zhang et al. [19] investigated the use of pre-trained models, such as BART, for generating summaries of pull request titles in GitHub projects, achieving high accuracy. Mastropaolo et al. [18] explored the use of transfer learning for code-related tasks, highlighting the adaptability of pre-trained models across different programming contexts. These studies highlight the potential of pre-trained models in improving code quality and understanding. However, they primarily focus on tasks like code review and summarization, leaving a gap in the exploration of code smell detection. Our study complements these works by applying pre-trained models, specifically Google Gemini, to the task of identifying code smells, providing insights into their effectiveness in this domain.

Prompt engineering has emerged as a key technique for optimizing the performance of LLMs in software engineering tasks. Shin et al. [12] analyzed different prompt engineering approaches for tasks, such as code generation and translation, finding that conversation-structured prompts yielded the best results. Similarly, White et al. [21] applied prompt design strategies to tasks like requirements elicitation and testing, demonstrating the importance of well-crafted prompts in achieving accurate outputs. Liu et al. [16] conducted a systematic survey of prompting methods in NLP, emphasizing the role of prompt engineering in enhancing model performance. While these studies focus on general software engineering tasks, our work specifically investigates the role of prompt engineering in code smell detection. We compare the effectiveness of generic and detailed prompts in guiding Google Gemini to identify code smells, contributing to the understanding of how prompt design influences the performance of LLMs in this context.

Recent studies have begun to explore the use of LLMs for code smell detection. For example, Alshahwan et al. [13] developed a tool based on LLMs to improve human-written tests, achieving a 73% acceptance rate for suggestions. Imai [20] evaluated GitHub Copilot's performance in collaborative programming, noting its ability to generate code but also highlighting quality concerns. Rane et al. [15] compared the capabilities of Google Gemini and ChatGPT, providing insights into their strengths and limitations in various tasks, including code analysis. While these studies demonstrate the potential of LLMs in improving code quality, they do not specifically address code smell detection. Our work builds on these foundations by evaluating Google Gemini's ability to detect code smells in Java projects, and exploring the impact of different prompt strategies. This focus on code smell detection fills a gap in the literature and provides new insights into the application of LLMs for software quality improvement.

Nunes et al. [10] evaluated the effectiveness of LLMs, specifically Copilot Chat and Llama 3.1, in fixing maintainability issues in real-world Java projects. Their study employed zero-shot and few-shot prompting techniques, finding that Llama with few-shot prompting successfully fixed 44.9% of the methods, while Copilot Chat and Llama zero-shot fixed 32.29% and 30%, respectively. However, the study also highlighted that most solutions introduced errors or new maintainability issues, emphasizing the need for human oversight. This work aligns with our research by providing empirical evidence of the challenges and potential of LLMs in addressing maintainability issues. While Nunes et al. [10] focus on fixing maintainability issues, our study extends this by specifically targeting code smell detection, providing a complementary perspective on the capabilities and limitations of LLMs in software maintenance tasks.

Our study is inspired by the work of Silva et al. [2], which investigated the use of ChatGPT for detecting code smells. While their study provided initial insights into the capabilities of LLMs for this task, our work extends their findings by evaluating Google Gemini, a more recent and advanced model. By replicating their methodology, we aim to provide a deeper understanding of how different LLMs perform in code smell detection and how prompt strategies influence their effectiveness. This replication and extension contribute to the growing body of research on the application of LLMs in software engineering.

VII. Conclusion and Future Work

This study explored Google Gemini's ability to detect code smells in Java projects in a dataset containing four types of smells: Blob, Data Class, Feature Envy, and Long Method. The results indicated that Gemini, using generic and detailed prompts, can identify the evaluated code smells, even detecting smells previously unidentified in the original dataset (which may or may not be a false positive). The study suggests that

Gemini could be a promising tool for helping developers identify code problems, paving the way for future research into the optimization and application of the model in different software development contexts. However, further research is still needed to determine Gemini's accuracy in detecting code smells and the influence of the type of prompts on its performance.

As part of future work, we propose a more in-depth analysis of Gemini's calibration, since for this research we used the model's default settings. We intend to explore different configuration parameters, assessing whether specific adjustments can have a positive impact on the results, both in terms of the accuracy and consistency of code smells detections. This research will include systematic experiments to identify the ideal calibration for the types of analysis carried out, based on metrics, such as the number of correctly identified code smells and the reduction in processing errors.

Another point to be investigated is why Gemini detected additional code smells that were not cataloged in the original dataset. We plan to conduct a detailed manual review of the dataset used, with a view to identifying possible flaws or gaps in the initial catalog of code smells. This may include a manual and semi-automatic re-analysis of the code contained in the repositories to check whether these additional problems detected by Gemini really reflect previously overlooked code smells or whether they stem from errors in the model.

In addition, we propose to investigate Gemini's accuracy in categorizing code smells using detailed prompts. This study would compare the results obtained between generic and detailed prompts, identifying how the structure of the prompt influences the model's ability to detect and classify code smells with greater accuracy. We also intend to calculate the precision and recall rate for the different types of prompts, enabling a more robust quantitative assessment of Gemini's effectiveness in these scenarios. Finally, we aim to further evaluate Gemini's effectiveness in other datasets of code smells, in order to verify the generalizability of the results. This will make it possible not only to validate the findings of this study, but also to understand how the model behaves in the face of code from a variety of contexts and development styles.

Acknowledgements. This research was partially supported by Brazilian funding agencies: CNPq (Grant 312920/2021-0), CAPES, and FAPEMIG (Grant APQ-01488-24).

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[2] L. Silva, J. Silva, J. Montandon, M. Andrade, and M. T. Valente, "Detecting Code Smells using ChatGPT: Initial Insights". Int'l Symposium on Empirical Software Engineering and Measurement (ESEM), 2024.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All you Need. In Advances in Neural Information Processing Systems", vol. 30, Curran Associates.

[4] K. Papapanos and J. Pfeifer, "A Literature Review on the Impact of Artificial Intelligence in Requirements Elicigtation and Analysis", Department of Computer and Systems Sciences Master level, 2023.

[5] K. Ahmada, M. Abdelrazeka, C. Arorac, M. Banob, J. Grundyc, "Requirements Practices and Gaps When Engineering Human-Centered Artificial Intelligence Systems", Elsevier Science Publishers, 2023.

[6] A. Santana, E. Figueiredo and J. Pereira, "Unraveling the Impact of Code Smell Agglomerations on Code Stability," International Conference on Software Maintenance and Evolution (ICSME), 2024.

[7] Pooja S, Chandrakala C. B. and L. K Raju, "Developer's Roadmap to Design Software Vulnerability Detection Model Using Different AI Approaches," IEEE Access, vol. 10, pp. 75637-75656, 2022.

[8] L. Madeyski and T. Lewowski, "MLCQ: Industry-Relevant Code Smell Data Set",Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering, 2020.

[9] N. Rane et al., "Gemini Versus ChatGPT: Applications, Performance, Architecture, Capabilities, and Implementation", 2024.

[10] H. Nunes, E. Figueiredo, L. Rocha, S. Nadi, F. Ferreira, and G. Santos, "Evaluating the Effectiveness of LLMs in Fixing Maintainability Issues in Real-World Projects". International Conference on Software Analysis, Evolution and Reengineering (SANER), 2025.

[11] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task Learning based Pre-trained Language Model for Code Completion", 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020.

[12] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks", 2023.

[13] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated Unit Test Improvement using Large Language Models at Meta", International Conference on the Foundations of Software Engineering (FSE), 2024.

[14] M. Siddiq, J. Santos, R. Tanvir, N. Ulfat, F. Rifat, and V. Lopes, "Using Large Language Models to Generate JUnit Tests: An Empirical Study", Int'l Conf. on Evaluation and Assessment in Soft. Engineering, 2024.

[15] A. Mastropaolo, N. Cooper, D. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using Transfer Learning for Code-Related Tasks", IEEE Transactions on Software Engineering, 2022.

[16] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing", 2021.

[17] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using Pre-Trained Models to Boost Code Review Automation", International Conference on Software Engineering (ICSE), pp. 2291-2302, 2022.

[18] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards Automating Code Review Activities", International Conference on Software Engineering (ICSE), 2021.

[19] T. Zhang, I. Irsan, F. Thung, D. Han, D. Lo, and L. Jiang, "Automatic Pull Request Title Generation", International Conference on Software Maintenance and Evolution (ICSME), 2022.

[20] S. Imai, "Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study", International Conference on Software Engineering (ICSE-Companion), 2022.

[21] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. Schmidt, "ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design", 2023.

[22] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," International Conference on Evaluation and Assessment in Software Engineering (EASE), 2016.

[23] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells", IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014.

[24] M. Chen, J. Tworek, H. Jun, et al., "Evaluating Large Language Models Trained on Code", 2021.

[25] L. Madeyski and T. Lewowski. "MLCQ: Industry-Relevant Code Smell Data Set", Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering (EASE), 342–347, 2020.

[26] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms: An empirical study," International Conference on Technical Debt (TechDebt), 2020.

[27] C. Zhifei, C; Lin, M. Wanwangying, Z. Xiaoyu, Z. Yuming, and X. Baowen, "Understanding metric-based detectable smells in python software: A comparative study", Information and Software Technology, vol. 94, pp. 14–29, 2018.

[28] N. Cardozo, I. Dusparic, and C. Cabrera, "Prevalence of Code Smells in Reinforcement Learning Projects", International Conference on AI Engineering (CAIN), 37-42, 2023.