Comparing Structural Quality of Code Generated by LLMs: A Static Analysis of Code Smells

Eduardo Sousa[1], **Erlene Santos**[1], **Alberto Sampaio**[1], **Carla Bezerra**[1]

[1]Universidade Federal do Ceará (UFC)
Quixadá – CE – Brasil

{eduardo12,erlenesantos}@alu.ufc.br, {albertosampaio,carlailane}@ufc.br

Abstract. *The automation of code generation through Large Language Models (LLMs) has emerged as a promising approach to support software development. However, concerns remain regarding the structural quality of the code produced, particularly the presence of code smells that affect maintainability. This paper presents an empirical study comparing code smells in outputs from four major LLMs: ChatGPT, DeepSeek, Amazon CodeWhisperer, and GitHub Copilot. Our analysis of 64 code units, generated from open-source projects, revealed that 60.9% contained at least one code smell. The results show significant variation, with DeepSeek having the lowest incidence of smells (43.8%) and Amazon CodeWhisperer the highest (68.8%). Long Method was the most frequent smell, constituting 40% of all occurrences. These findings provide empirical evidence of the structural quality differences in LLM-generated code and highlight the need for rigorous, automated quality assurance.*

1. Introduction

Automated code generation has established itself as a promising strategy to support software development, especially in contexts of increasing complexity and need for productivity, since this practice aims to reduce human effort in the production of artifacts and, at the same time, maintain the quality of the generated code [Harman et al. 2015]. However, ensuring that the code produced automatically meets organization and maintenance standards remains a significant challenge, with so-called code smells, which consist of signs that the code has design deficiencies even if functional, being one of the main indicators of poor structural quality [Khomh et al. 2009a]. These problems do not cause direct execution failures, but indicate a propensity for future difficulties in understanding, modifying, and extending the system [Fowler 2018]. Studies show that such characteristics, by making it difficult to understand and change the code, directly impact the maintainability of the software and, consequently, make its life cycle more expensive and prone to errors. [Yamashita and Moonen 2012].

The need to reduce manual effort and optimize the development process has driven interest in automatic code generation approaches, since the adoption of these solutions seeks to minimize the time and costs of manual coding, expand functional coverage and improve the consistency of the produced artifacts [Danglot et al. 2019]. Furthermore, by reducing dependence on human intervention, automation contributes to the standardization of development practices, which can favor the technical quality of the code [Gvero et al. 2013]. In increasingly complex systems, these techniques become essential to ensure the efficiency of validation and maintenance processes, without compromising the reliability and quality of the products delivered [Briand et al. 2003].

Large-Scale Language Models (LLMs) have gained prominence in automatic code generation by translating natural language descriptions into functional software snippets [Chen et al. 2021]. Tools like GitHub Copilot[1], ChatGPT[2], Amazon CodeWhisperer[3] e DeepSeek[4] exemplify this trend, enabling the automation of programming tasks previously performed manually [Nijkamp et al. 2022]. These models demonstrate the ability to understand the semantic context of the code and generate solutions compatible with development standards, overcoming limitations of approaches based on fixed rules [Rabinovich et al. 2017]. However, despite the advances, there are still uncertainties regarding the structural quality of the codes produced, especially with regard to the presence of code smells, which reinforces the need for empirical studies on their impacts on software engineering [Palomba et al. 2014].

Automatic code generation by LLMs has been consolidated as a promising approach in software development. However, the structural quality of the artifacts generated by these tools still presents relevant challenges [Pearce et al. 2025]. Studies show that although solutions like DeepSeek and ChatGPT are capable of producing functional code, they often violate good design and maintenance practices, evidencing the presence of code smells and other structural deficiencies [Sadik and Govind 2025]. These signs of low architectural quality can directly impact the readability, maintainability and future evolution of systems, making it necessary to ensure that, despite advances in automation, the generated code presents structural standards compatible with those required in production environments [Tufano et al. 2017].

In this context, this work proposes an empirical study on the structural quality of codes generated by different LLMs, focusing on the detection of code smells. The objective is to compare the code production performed by models such as ChatGPT, DeepSeek, Amazon CodeWhisperer and GitHub Copilot, evaluating their adherence to good design practices. For this, open source projects available on GitHub[5] will be used, in which LLMs will generate code snippets from standardized prompts. Then, an automatic analysis will be performed with the Designite[6] tool to identify code smells present in the produced artifacts. This research provides critical empirical evidence on this matter, revealing that a significant portion (over 60%) of the code generated in our study contained structural flaws, thereby highlighting the current potential and limitations of adopting LLMs in software engineering.

2. Theoretical Basis

In this section, the main theoretical concepts that support this study are presented, addressing language models, code smells and static analysis.

[1]`https://github.com/features/copilot` Accessed on: April 27, 2025
[2]`https://chat.openai.com/` Accessed on: April 27, 2025
[3]`https://aws.amazon.com/pt/codewhisperer/` Accessed on: April 27, 2025
[4]`https://www.deepseek.com/` Accessed on: April 27, 2025
[5]`https://github.com/` Accessed on: April 27, 2025
[6]`https://www.designite-tools.com/` Accessed on: May 3, 2025

2.1. Large Language Models

LLMs are deep neural networks designed to process, understand, and generate natural language and source code with a high degree of coherence. They are trained on large textual corpora, which allows them to learn complex linguistic structures, programming patterns, contextual dependencies, and stylistic nuances. [Vaswani et al. 2017].

With the advancement of these models, it has become possible to apply them directly in development environments to support tasks such as code completion, function suggestions, restructuring logical blocks, and even generating entire system snippets. Their growing adoption is justified by the ability to accelerate development, reduce repetitive tasks, and assist in standardizing generated code. However, LLMs still face challenges like limited knowledge updates due to training cutoffs [Chen et al. 2021], along with frequent issues such as ambiguity, redundancy, and poor practices that compromise code quality. These factors emphasize the need for automated tools like Designite to detect code smells and structural weaknesses in cohesion, coupling, and organization.

This study evaluates four LLMs: GitHub Copilot [Chen et al. 2021], which provides real-time code suggestions through OpenAI's Codex models; ChatGPT [Achiam et al. 2023], known for generating structured code from natural language using GPT-4.5; Amazon CodeWhisperer [Pearce et al. 2025], focused on secure, context-aware generation within the AWS ecosystem; and DeepSeek [Sadik and Govind 2025], an emerging model noted for its precision and simplicity in code generation.

2.2. Code Smells

The internal quality of software depends on clarity, organization, and adherence to good development practices. *Code smells* are indicators of poor design that, although not causing immediate failures, compromise maintainability, extensibility, and comprehensibility over time [Fowler 2018, Marinescu 2004].

This study focuses on seven widely recognized code smells implemented in the Designite tool for Python: Long Method, Feature Envy, Long Parameter List, God Class, Data Class, Duplicated Code, and Complex Conditional. These represent violations of core object-oriented principles such as cohesion, low coupling, and encapsulation [Chidamber and Kemerer 1994].

For instance, Long Method impairs readability by grouping multiple logics [Fowler 2018]; Feature Envy arises when a method excessively accesses data from other classes [Fowler 2018]; Long Parameter List complicates methods with many arguments [Fowler 2018]; God Class and Data Class reflect poor responsibility distribution [Marinescu 2004]; Duplicated Code increases maintenance effort [Moha et al. 2009]; and Complex Conditional complicates decision-making logic [Fowler 2018].

These smells were selected for their practical relevance, prevalence in real systems, and direct impact on internal quality. Studies associate their presence with increased defect-proneness and reduced maintenance efficiency [Khomh et al. 2009b]. Their detection is effectively supported by static analysis tools like Designite, which apply structural metrics and heuristics [Moha et al. 2009]. Focusing on these categories, this study establishes a robust comparative analysis of code generated by language models, supporting the evaluation of structural quality and adherence to modern software engineering principles.

2.3. Static Analysis

Static analysis [Cousot and Cousot 1977] is a technique widely used in software engineering to inspect source code without having to execute it. This approach allows identifying structural problems, such as code smells, by applying metrics and heuristic rules directly to the program structure. Unlike dynamic analysis, which relies on the execution of the system, static analysis provides an objective view of the internal quality of the software at development time.

Tools like Designite stand out in this scenario by automating the detection of smells based on characteristics such as complexity, cohesion and coupling [Moha et al. 2009]. Such tools produce detailed reports that help quantify structural degradation and compare quality between different versions of code, such as those generated by language models.

In addition to providing consistent and reproducible diagnostics, static analysis is especially useful in comparative studies, allowing the evaluation of the structural robustness of the artifacts produced without the influence of external factors such as the execution environment or input data. Its application in this work allows an objective analysis of the quality of the code generated by LLMs.

3. Related Work

[Siddiq et al. 2022] established that LLMs like GitHub Copilot do indeed generate code with smells. Their study applied static analysis to code generated by transformer-based models and found a high frequency of code smells such as Long Method and God Class, emphasizing the need for quality checks even when code is syntactically correct.

Other work has focused on the reverse task: [Zhang et al. 2024] demonstrated the potential of Copilot to correct code smells. Their approach showed partial success in refactoring issues like Long Method and Duplicated Code, though deeper structural problems, such as Feature Envy, remained challenging.

[Silva et al. 2024] and [Sadik and Govind 2025] evaluated the ability of ChatGPT and DeepSeek to detect code smells. [Silva et al. 2024] explored ChatGPT explanations based on smell types, while [Sadik and Govind 2025] showed that DeepSeek could identify common structural flaws with reasonable accuracy when guided by well-formed prompts.

More recently, [Velasco et al. 2024] introduced the CodeSmellEval framework to measure how prone LLMs are to generating code smells. The benchmark assesses smell density and diversity across models, offering a foundation for comparative analysis like the one presented in this study.

Unlike these works, our study performs a comparative analysis focused on the generation of code smells by four distinct language models. By using a unified static analysis methodology, this paper provides a direct overview of the structural quality of the code produced by each tool under the same conditions.

4. Methodology

This study aims to assess the structural quality of code generated by LLMs by identifying code smells using the Designite tool. The evaluation compares models based on the frequency and types of detected smells, code complexity, and overall performance. Python was selected for its widespread use and compatibility with the tools. Figure 1 illustrates the full methodological process, including data selection, generation, and analysis steps.

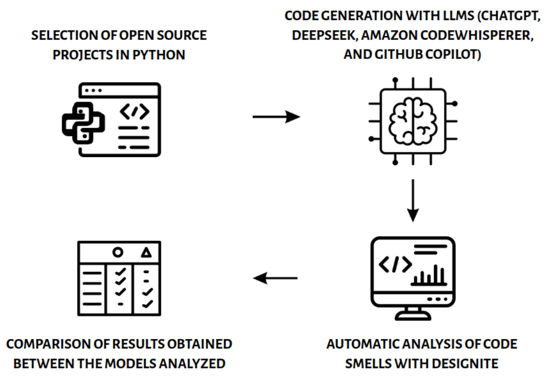
SELECTION OF OPEN SOURCE PROJECTS IN PYTHON
CODE GENERATION WITH LLMS (CHATGPT, DEEPSEEK, AMAZON CODEWHISPERER, AND GITHUB COPILOT)
AUTOMATIC ANALYSIS OF CODE SMELLS WITH DESIGNITE
COMPARISON OF RESULTS OBTAINED BETWEEN THE MODELS ANALYZED

Figure 1. Methodological procedures adopted in the study.

4.1. Project Selection

The first step of this study was selecting four open-source Python repositories from GitHub. Selection criteria included well-defined functionalities, adherence to recognized coding standards, simplicity in dependency structures, and clear package organization. Additionally, aspects such as code readability and separation between logic layers were considered to enable a more accurate and reliable assessment of structural quality. The selection involved GitHub searches using terms such as “python validation library” and “data analysis tool”, followed by manual curation based on the outlined criteria.

In addition to these criteria, priority was given to selecting projects that presented production code with low coupling between modules, which facilitates the punctual analysis of the structures generated by the models. This approach helps to reduce interference between different parts of the system and allows for a more effective application of the Designite tool in detecting code smells [Alfadel et al. 2020].

For each selected project, production files with representative methods and classes were identified. These snippets served as the basis for the automated generation of new code versions through LLMs. The data preparation and organization process was carried out separately for each project, in order to avoid redundancies, mitigate noise and reduce the risk of hallucinations during the generation stage. Furthermore, we sought to preserve the cohesion between the extracted elements and ensure that the selected fragments were sufficiently complete to allow an accurate and functional reconstruction of the code by the models, respecting the original implementation context.

Table 1 lists the projects and their main characteristics.

4.2. Code Generation with LLMs

With the projects defined, the next step involved automated code generation using four language models: **ChatGPT** (GPT-4.5), **DeepSeek** (DeepSeek-V3-0324), **Amazon CodeWhisperer** (integrated into Amazon Q Developer, April 2024 release), and **GitHub Copilot** (with coding agent, May 2025 update). In total, 16 different sets of codes were generated, applying the four models to each of the four projects.

Table 1. Characteristics of the projects used in the evaluation

Project	Domain	LOC	Classes	Functions
voluptuous	Data Validation	3.875	80	348
pandas-profiling	Exploratory Analysis	22.444	151	873
loguru	Log Record	19.162	100	1.397
pydantic-core	Structural Validation	43.766	572	2.275

The instructions provided to the models were composed of snippets of classes and methods extracted directly from the original repositories. Each generation command was built in a standardized manner, guiding the models to rewrite the codes while maintaining the original logic, with clear definition of the scope and without ambiguity. The goal was to preserve the expected functionality while analyzing the syntactic and organizational structure of the generated code.

To standardize the input provided to the models and ensure consistency between experiments, a template with clear and specific instructions was adopted. The prompt structure was adapted according to the context of each project, respecting the nomenclature, class structure and single responsibility principles. Below is a generic example of the instruction used:

> **Prompt used:** Generate a new version of this code in Python. Keep the same functionality, respecting the names of classes, methods and attributes. Reorganize the structure if necessary, aiming to improve readability and reduce complexity. Eliminate unnecessary statements and avoid redundant comments.
> Class: `UserValidator`
> Description: Class responsible for validating user input data in a registration system.

```
class UserValidator:
    def validate_email(self, email):
        if '@' not in email or '.' not in email:
            raise ValueError('Invalid␣email')
```

This approach allowed the process to be conducted in a controlled manner, ensuring that all models received equivalent commands in terms of content and scope. Generation was performed separately for each model and project, avoiding any overlapping of information. After generation, all files produced were automatically validated by running them in a Python 3.11 environment. Only codes that presented valid syntactic behavior were submitted for subsequent analysis. At the end of this stage, 64 code units were collected, representing the structural variations produced by the models on the same input contexts.

4.3. Code Smell Detection with Designite

With the codes generated by the language models properly organized, the structural analysis stage began using the Designite tool, version for Python. This tool was used for its ability to automatically detect a wide variety of code smells related to problems of cohesion, complexity, coupling and poor design. The version used was DesignitePython v.1.4.0, executed locally in a controlled development environment.

Each of the files generated by the LLMs were individually subjected to the static inspection process. The tool was configured to perform a complete scan of each directory corresponding to a model project, resulting in reports containing the types and quantities of code smells identified.

To ensure uniformity of analysis, all directories were organized with the same structure, containing exclusively the files generated by each LLM, without interference from other auxiliary modules. The output produced by Designite was converted into structured spreadsheets with the results grouped by project and model. This consolidation allowed direct comparison of the structural quality of the code between the different LLMs applied to the same context.

In addition to the absolute count of smells, the diversity of detected types was recorded, allowing us to assess not only the frequency but also the variety of structural problems introduced by each model. This approach provided a comprehensive view of the recurring patterns of fragility in the automatically generated code, serving as the basis for the results discussed in the following section.

5. Results and Discussion

This section presents results from the automated analysis of code smells and structural complexity in the generated code units. It covers the frequency and types of identified smells, complexity metrics, and differences among the models.

5.1. Frequency of code smell detection and structural complexity

The analysis performed with the Designite tool allowed us to identify the presence of code smells in the code units generated by the evaluated language models. In total, 64 code units were analyzed (16 per model), of which 39 units presented at least one code smell, corresponding to 60.9% of the total.

Table 2 presents the distribution of code smell frequency by model, classifying units according to the number of code smells detected: one, two, three, or more. It also includes the number of units in which no code smells were identified, as well as the overall frequency of units affected by code smells for each model.

Table 2. Frequency of code smells by model

Model	1 smell	2 smells	$\geq$3 smells	No smells	Percentage
ChatGPT	7	2	1	6	62,5%
DeepSeek	6	1	0	9	43,8%
Amazon CodeWhisperer	5	4	2	5	68,8%
GitHub Copilot	7	3	1	5	68,8%
Total	**25**	**10**	**4**	**39**	**60,9%**

It can be seen that, although all models generated units containing code smells, the proportion of affected units varies between them. Amazon CodeWhisperer and GitHub Copilot presented the highest proportions of units with code smells (68.8%), while DeepSeek was the model with the lowest incidence (43.8%). In addition to frequency, the structural complexity of the code units with code smells was also measured, using the Average Cyclomatic Complexity [Ebert et al. 2016].

Table 3 shows how the models differ in structural complexity.

Table 3. Average cyclomatic complexity by model

Model	Average Cyclomatic Complexity
ChatGPT	5,2
DeepSeek	3,8
Amazon CodeWhisperer	7,6
GitHub Copilot	6,9

The results indicate that the models that presented a higher frequency of code smells also tend to generate code with greater structural complexity. Amazon CodeWhisperer presented the highest average cyclomatic complexity (7.6), followed by GitHub Copilot (6.9). On the other hand, DeepSeek again stood out for presenting not only the lowest frequency of code smells, but also the lowest average complexity (3.8), indicating a tendency to produce structurally simpler code.

These findings reinforce the existence of a correlation between code complexity and the presence of code smells, highlighting the need to consider not only functionality, but also structural quality when using language models for automatic code generation.

5.2. Distribution of identified code smells

The occurrences of code smells were counted according to their types, based on the analysis of the generated code units. Table 4 presents the distribution of these occurrences, indicating both the absolute quantity and the relative proportion of each type of smell in relation to the total detected, as well as its frequency in relation to the total of 64 units analyzed.

Table 4. Code smell occurrences by type

Code Smells	Occurrences	% Occurrences	% Frequency
Long Method	30	40%	46,9%
Feature Envy	15	20%	23,4%
Long Parameter List	8	10,7%	12,5%
God Class	7	9,3%	10,9%
Data Class	6	8%	9,4%
Duplicated Code	5	6,7%	7,8%
Complex Conditional	4	5,3%	6,3%
Total	**75**	**100%**	—

Long Method was the most frequently identified type of code smell, with 30 occurrences, representing 40% of the total smells detected and appearing in approximately 47% of the units analyzed. Next, Feature Envy stands out, with 15 occurrences (20%), often associated with the excessive use of data from other classes, which compromises cohesion and increases coupling.

Other smells, such as Long Parameter List and God Class, were also detected with relative frequency, indicating the presence of methods with excessive parameters and classes that concentrate multiple responsibilities, respectively.

The Data Class and Duplicated Code smells appeared on a smaller scale, but still point to important issues related to redundancy and inadequate encapsulation. Finally, Complex Conditional was the least frequent, with 4 occurrences, but deserves attention due to its impact on code readability and maintainability. The results indicate that structural issues mainly relate to poor architectural practices like long methods, excessive coupling, and low cohesion, which hinder system maintainability and evolution.

5.3. Comparative analysis between LLMs

The comparative analysis reveals clear differences in structural quality, code smell frequency, and complexity between the models. As shown in Figure 2, Amazon CodeWhisperer presented the highest number of code smells, especially Long Method (12) and Complex Conditional (3), consistent with its higher average cyclomatic complexity (7.6) and reinforcing the link between complexity and code smells.

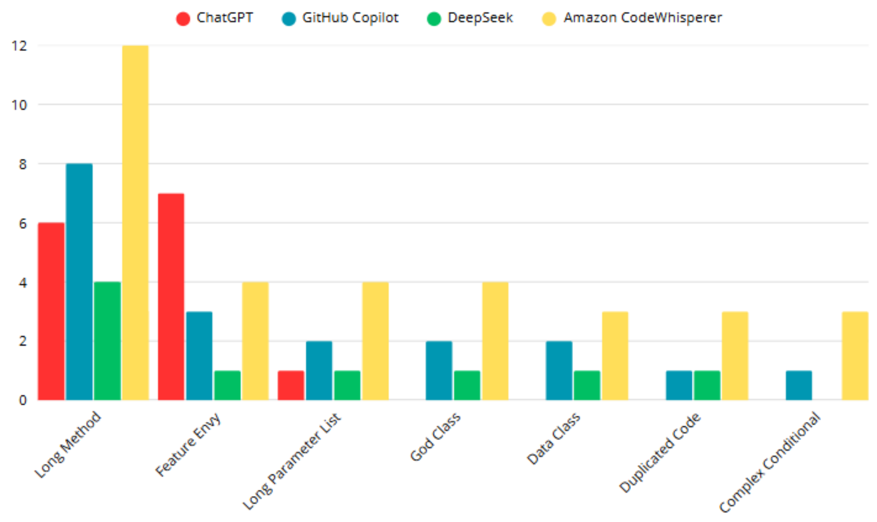

Figure 2. Distribution of code smell types by language model

Similarly, GitHub Copilot showed high frequencies of several code smells, especially Long Method (8 occurrences) and Duplicated Code (1 occurrence). Its average cyclomatic complexity of 6.9 reinforces this trend, as demonstrated by previous studies that associate more complex codes with greater susceptibility to the emergence of smells, especially those related to method extension and conditional complexity [Fowler 2018].

ChatGPT, in turn, presented a distinct profile, with a moderate overall incidence of code smells (10 affected units, 62.5%), but a notable concentration of Feature Envy instances (7 of the 15 total occurrences). This pattern indicates a tendency of the model to generate methods that rely excessively on data from other classes, a structural weakness that impairs cohesion and is considered a widely recognized anti-pattern in object-oriented design [Chidamber and Kemerer 1994]. Despite presenting an intermediate average cyclomatic complexity (5.2), the code generated by ChatGPT reflects an architectural predisposition that is different from that of the other models.

The DeepSeek model stands out for its low incidence of code smells (7 units, 43.8%) and lower cyclomatic complexity (3.8), indicating a tendency to generate simpler, more cohesive code. Its limited occurrences of Long Method (4 of 30) and Feature Envy (1 of 15) support this. Previous studies link reduced complexity to better maintainability and fewer code smells [Marinescu 2004], suggesting DeepSeek favors cleaner code.

Figure 2 shows that Amazon CodeWhisperer and GitHub Copilot contribute most to the diversity of code smells, appearing in all seven categories. This suggests that, despite strong generation capabilities, they are more prone to structural issues in the absence of proper quality control [Moha et al. 2009]. In contrast, DeepSeek presents minimal presence across most smells, while ChatGPT concentrates its weaknesses on cohesion-related problems.

Higher cyclomatic complexity correlates with more code smells [Tufano et al. 2017], highlighting the value of static analysis tools like Designite. Overall, CodeWhisperer and Copilot produce more complex and smell-prone code. DeepSeek generates simpler outputs, while ChatGPT requires attention to cohesion due to Feature Envy.

5.4. Practical Implications

The results of this study help guide the selection of LLMs and improve the efficiency and reliability of code review processes. In addition to providing insights into their practical use, the findings provide support for refining these models and understanding their limitations. The work also sets a benchmark for future research and highlights the importance of rigorously assessing the quality of AI-generated code, which should not be accepted without proper validation and human oversight. This supports the responsible and informed integration of AI tools into software development workflows.

6. Limitations and Threats to Validity

This study has limitations that are worth noting. The analysis was restricted to four Python projects, which may not represent other software contexts. The detection of code smells relied on a single static analysis tool (Designite), and the results could vary with the use of other tools or a different set of smells. In addition, the performance of the LLMs was evaluated based on a standardized prompt and specific versions of the models, and future updates or different prompting approaches may alter the results. As threats to validity, we can cite the small sample size, limiting the generalizability of the results and the analysis depended on one tool.

7. Conclusion and Future Work

This study's analysis of four LLMs revealed significant differences in structural quality, with 60.9% of 64 generated units containing code smells. DeepSeek produced the highest quality code with the lowest smell incidence (43.8%), while Amazon CodeWhisperer and GitHub Copilot had the highest (68.8%). ChatGPT showed an intermediate profile with specific cohesion issues related to Feature Envy. These findings highlight the importance of applying automated static analysis when using LLMs for code generation. As future work, we plan to extend the analysis to additional models and programming languages, as well as explore strategies to improve the structural quality of generated code.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Alfadel, M., Aljasser, K., and Alshayeb, M. (2020). Empirical study of the relationship between design patterns and code smells. *Plos one*, 15(4):e0231731.

Briand, L. C., Labiche, Y., and O'Sullivan, L. (2003). Impact analysis and change management of uml models. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 256–265. IEEE.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252.

Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., and Baudry, B. (2019). A snowballing literature study on test amplification. *Journal of Systems and Software*, 157:110398.

Ebert, C., Cain, J., Antoniol, G., Counsell, S., and Laplante, P. (2016). Cyclomatic complexity. *IEEE software*, 33(6):27–29.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Gvero, T., Kuncak, V., Kuraj, I., and Piskac, R. (2013). Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 27–38.

Harman, M., Jia, Y., and Zhang, Y. (2015). Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE.

Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009a). An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., and Sahraoui, H. (2009b). A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. (2022). Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2014). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2025). Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2):96–105.

Rabinovich, M., Stern, M., and Klein, D. (2017). Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*.

Sadik, A. R. and Govind, S. (2025). Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3. *arXiv preprint arXiv:2504.16027*.

Siddiq, M. L., Majumder, S. H., Mim, M. R., Jajodia, S., and Santos, J. C. (2022). An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–82. IEEE.

Silva, L. L., Silva, J. R. d., Montandon, J. E., Andrade, M., and Valente, M. T. (2024). Detecting code smells using chatgpt: Initial insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 400–406.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2017). When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Velasco, A., Rodriguez-Cardenas, D., Alif, L. R., Palacio, D. N., and Poshyvanyk, D. (2024). How propense are large language models at producing code smells? a benchmarking study. *arXiv preprint arXiv:2412.18989*.

Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE.

Zhang, B., Liang, P., Feng, Q., Fu, Y., and Li, Z. (2024). Copilot-in-the-loop: Fixing code smells in copilot-generated python code using copilot. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2230–2234.