CASE STUDY: PROTOTYPE OF A CODE REVIEW SYSTEM FOR DETECTING AND SUGGESTING CORRECTION OF CODE SMELLS WITH THE AID OF ARTIFICIAL INTELLIGENCE

Cláudio Ratke, Cairo Augusto de Andrade, Djonathan Mariano D'Avila, Edson Eurides Adriano, Eduardo Roberto Schulle and Jonathan Bauer[1]

[1]Faculdade Senac Blumenau, Blumenau, Santa Catarina, Brasil
claudio.ratke@sc.senac.br

ABSTRACT

The code review process is essential to ensure the quality, maintainability, and sustainable evolution of software projects. However, the high demand for reviews, coupled with the increasing complexity of systems, can lead to superficial or inconsistent evaluations. In this regard, it was developed using artificial intelligence to analyse the latest updates of source code files in repositories (pull requests), identifying potential code smells and suggesting improvements. The project consists of creating an application capable of evaluating the code added and modified in the pull request, providing reports on detected code smells such as long methods, code duplications, and classes with multiple responsibilities. By applying natural language processing and prompt engineering techniques, the system recognizes bad practice patterns and proposes refactoring aligned with widely accepted software engineering guidelines. Furthermore, the tool standardizes the feedback offered to developers, reducing the subjectivity typically found in manual reviews. Artificial Intelligence plays a key role in this process by offering advanced contextual analysis and suggestion generation, ensuring a more comprehensive and agile approach to problem detection. In summary, the prototype represents a solution aimed at enhancing the code review process, contributing to improved software quality and optimization of the time spent on refactoring.

KEYWORDS

Code Review, Code Smells, Artificial Intelligence

1. INTRODUCTION

Modern software development has widely adopted agile methodologies, which operate in iterative and incremental delivery cycles called sprints, in which functionalities are continuously planned, implemented, and tested [1]. This process is guided by user stories, allowing for frequent value deliveries through collaboration between team members [2].

Among the essential practices in this context, code review stands out, which consists of the collaborative analysis of the source code to ensure quality, adherence to good practices, and readability [3],[4]. In addition to promoting technical improvements, the review fosters knowledge sharing and professional growth within the team [5].

However, as projects grow and complexity, manual review becomes more challenging, especially given the high volume of pull requests and limited reviewers' time [6]. Among the recurring problems in this process are code smells — signs of poor structure in the code that, although they do not represent direct errors, indicate potential future difficulties, such as poor cohesion, high coupling, and poor readability [3], [4].

In addition, good programming practices, such as those advocated by [7] and [8], promote readability, organization, and efficiency. Principles such as meaningful names, small functions, and non-duplicate code are essential strategies for building sustainable systems.

On the other hand, code smells appear, defined by Beck and Fowler as indicators that a code may be poorly designed, even if it works correctly [3], [4]. Examples include long methods, classes with multiple responsibilities (God Class), excessive commenting, and code duplication. The presence of these smells negatively affects the maintainability, understanding, and evolution of the system [9].

The accumulation of code smells can lead to bugs. From this perspective, a systematic review of the literature carried out by [10] analysed 18 studies and showed that the presence of code smells is directly related to the increase in the incidence of bugs in software development. Of the studies analysed, 16 points to this correlation significantly. These bugs lead to a variety of losses for organizations, including high maintenance costs, reduced productivity for development teams, and potential revenue losses.

Furthermore, recent studies indicate that the cost of correcting defects increases significantly as software progresses through the phases of the lifecycle. For example, the IBM Systems Sciences Institute [11] reported that correcting a bug found after the product is released can be up to 100 times more expensive than if it were identified during the design phase.

These issues can compromise the scalability of the application, making adapting to new technologies or market requirements an expensive and time-consuming challenge [12].

In view of this, according to research conducted by [13], the removal of code smell occurrences in industrial systems can significantly improve the internal quality attributes of the software. In his study, which analysed several projects, the elimination of these anomalies showed notable gains in metrics such as cohesion, reduction of coupling and reduction of code complexity, as can be seen in Figure 1 (Impact of removing the occurrences of code smells on quality).

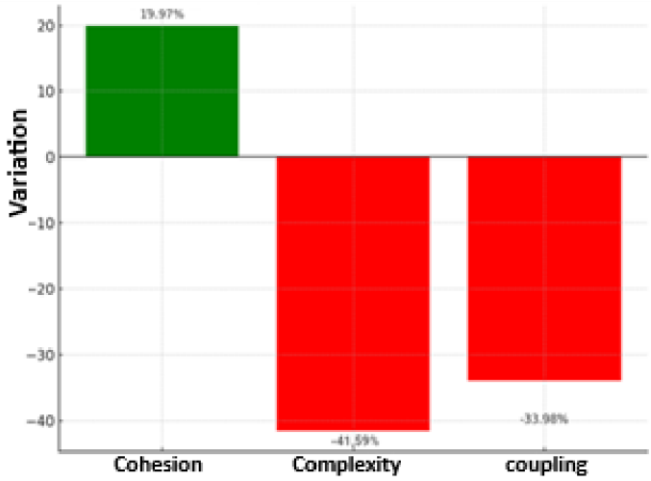

Figure 1 – Impact of removing the occurrences of code smells on quality.

1.1. CODE REVIEW

The code review process consists of analyzing the source code produced by a developer by another member of the team, aiming to identify errors, suggest improvements, and ensure adherence to the project's technical and style standards. This practice plays a key role in ensuring software quality, contributing to the early detection of failures and strengthening collaboration among team members [6].

The adoption of good programming practices is essential for the development of quality software. These practices aim to make code more readable, maintainable, and efficient, facilitating collaboration between developers and the continuous evolution of systems. Robert C. Martin, in The Clean Coder [7], points out that the developer's professional responsibility includes writing clean code. He argues that a clear and well-structured code facilitates communication between team members and reduces the incidence of errors. As can be seen in Table 1.

Table 1. Results of Evaluation pf Detection and Suggestion of Code Smell

Principle	**Description**
Meaningful Names	Use clear and descriptive names for variables, functions, and classes.
Small Functions	Maintain short roles that accomplish a single task.
Avoid duplication	Reuse code to prevent redundancies.
Error Handling	Manage exceptions clearly and consistently.
Justified Comment	Comment only when necessary to clarify the code.

Tom Hombergs, in turn, addresses the practical application of the principles of clean architecture. It demonstrates how the separation of responsibilities and the organization of code into well-defined layers contribute to the maintainability and scalability of software. [8] reinforces that the adoption of clean architecture is not only a technical issue but also strategic for the long-term success of projects.

The implementation of these practices directly impacts the productivity of development teams. Well-structured code reduces the time required for understanding and modification, allowing developers to focus on more complex and innovative tasks. Consequently, this results in faster deliveries and less likelihood of errors.

In contrast to good software development practices, the socalled "code smells" arise. The term "code smell" was first introduced by Kent Beck [4] and later popularized by Martin Fowler as a powerful metaphor to indicate subtle signs that a piece of code may be poorly designed or that, over time, it may become a source of more serious problems, even if it apparently still works correctly.

These "smells" are not necessarily functional failures or explicit bugs, but rather indications that suggest a possible deterioration of the internal structure of the system. They often violate fundamental principles of software design, such as cohesion between components, loose coupling between modules, and the readability and clarity of the source code. Thus, code smells

act as alerts that developers should seriously consider during the process of maintaining and continuously evolving software systems [4].

There are several types of code smells widely discussed and documented in the specialized literature on software engineering, each with specific characteristics and distinct impacts on the maintainability of the system. Among the examples most often found in real projects are the excessively long methods (Long Method), which concentrate too many responsibilities in a single block of code and make it difficult to understand and reuse.

In addition, evidence shows that the presence of code smells is strongly associated with architectural degradation and modularity violations, hindering the maintenance of critical applications [14]. Detecting these signals is not always trivial, as it requires careful analysis and often the support of automated tools to identify subtle patterns in the code.

Over the years, the cataloguing of code smells has expanded. Initially, Fowler [4] proposed a catalog with 22 types, but new categories were later introduced for several languages besides Java, such as MATLAB and Python [15].

Among the most recent approaches, the taxonomy proposed by [16] stands out, which groups code smells into the following categories:

- Bloaters: Components that have grown excessively large or complex.
- Object-Orientation Abusers: Violations of the fundamental principles of object-oriented programming.
- Change Preventers: Structures that make it difficult to safely modify software.
- Dispensable: Unnecessary elements that compromise the clarity of the code.
- Couplers: Excessive interdependence between classes or modules.

The following is a comprehensive selection of these smells, their descriptions, and associated categories, which can be seen in Table 2, based on Martin Fowler's catalogue in Refactoring: Improving the Design of Existing Code and recent studies [4].

Table 2. Catalog of Code Smells

Code Name Smell	**Description**
Divergent Change	A single class needs to be modified by different types of changes, indicating that it has multiple responsibilities.
Shotgun Surgery	A single change requires modifications to several different classes, showing that the logic is spread.
Long Parameter List	Methods or constructors with many parameters, making the code difficult to read and prone to errors.
Feature Envy	A method that accesses data from another class frequently, suggesting that it may be in the wrong class.
Data Clumps	Groups of data that appear together in multiple places, suggesting that they should be encapsulated in a class of their own.
Large Class/God	Classes that accumulate many responsibilities, attributes and methods. The

Class	excess of features makes it difficult to understand and maintain.
Long Method	Methods that are too long make the code difficult to understand and maintain. The ideal is to divide it into smaller and more specific methods.
Message Chains	Long sequences of cascading method call, making it difficult to understand and maintain.
Middleman	Classes that only pass calls to other objects, without adding value, can be eliminated.
Speculative Generality	Generic or abstract code with no real need, created in anticipation of future demands.
Refused Bequest	Code created for possible future needs that never materialize, making the system more complex unnecessarily.
Parallel Inheritance Hierarchies	When for each class in a hierarchy it is necessary to create a corresponding one in another hierarchy, leading to duplication of structures.
Alternative Classes with Different Interfaces	Classes that do similar things but have different interfaces, making it difficult to use interchangeably.
Inappropriate Intimacy	Classes that access internal details of other classes, violating encapsulation.
Comments	Overuse of comments to explain confusing code, rather than improving the clarity of the code itself.
Lazy Class	Classes that do not justify their existence because they have little functionality; can be removed or merged.
Duplicated Code	Identical or very similar code snippets appear in more than one place. This makes maintenance is difficult because a change needs to be replicated at multiple points.
Temporary Field	Fields in a class that are only used in specific situations, indicating that they could be moved to another structure.
Data Class	Classes that only store data, with no associated behavior, usually only with getters and setters.
Switch Statements	Frequent use of nested switch/case or if/else structures, which may indicate the need for polymorphism.
Primitive Obsession	Overuse of primitive types instead of creating more expressive types or classes for the domain.
Incomplete Library Class	Library classes that don't provide all the functionality you need, leading to alternative implementations or subclasses.

The presence of code smells does not indicate immediate errors, but it points to weaknesses in the design of the software that can hinder its evolution. As he points out [4], identifying these signs is essential to avoid future problems and maintain code quality. Refactoring to eliminate code smells makes the system clearer, more cohesive, and easier to maintain, contributing to its long-term sustainability.

2. RESEARCH

Data collection was carried out through the application of a questionnaire designed to obtain information about the use and importance of a code smell detection tool. The questionnaire was answered anonymously by the participants and contains 15 questions, 2 of which are multiple-choice and 13 with single-choice answers.

To identify the most relevant code smells in the perception of developers, an empirical survey was carried out with professionals in the area. Participants evaluated the impact of various smells in terms of compromising readability, maintainability, and code evolution. Figure 2 (Research Chart: Average importance of the top 10 code smells) presents the ten smells considered most critical. Among them, the repetition of code blocks, classes with multiple responsibilities and the need for scattered changes stand out, all with an average importance higher than 2.4 on a scale of 1 (low) to 3 (high). These results show the practical emphasis on aspects related to duplication, cohesion and coupling, corroborating studies such as those of [4] on the smells that most affect maintainability.

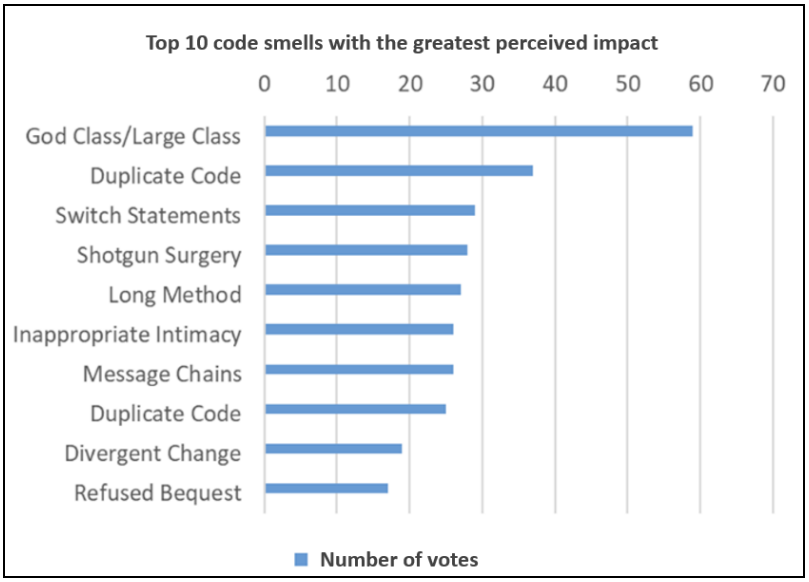

Figure 2. Average importance of the top 10 code smells.

3. PROTOTYPE PROPOSITION

To enable automatic review of pull requests, the prototype establishes direct integration with GitHub. This integration occurs through different features made available by the platform, such as authentication via OAuth 2.0, configuration of webhooks, and installation of a custom GitHub App, that is, automated agents developed to perform specific tasks within repositories.

The prototype architecture was structured in a modular way, using modern technologies and cloud services to ensure efficient integration with versioning platforms and ease of maintenance (Figure 3).

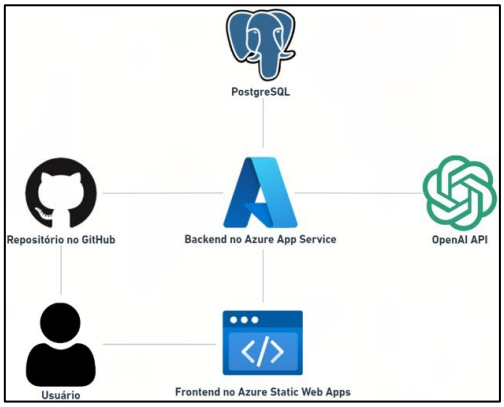

Figure 3. Prototype architecture.

Once installed, the GitHub app is integrated with GitHub webhooks. Webhooks are responsible for notifying the system whenever a new pull request is opened, updated, or reopened. These notifications are sent through HTTP requests containing the data necessary for the system to perform automated code analysis.

The prototype is designed to fully support the automated code review flow, as seen in Figure 4, especially focused on the analysis of pull requests and detection of code smells. The system includes entities that represent users, repositories, analysis rules, pull requests, code suggestions, and reports generated by artificial intelligence.

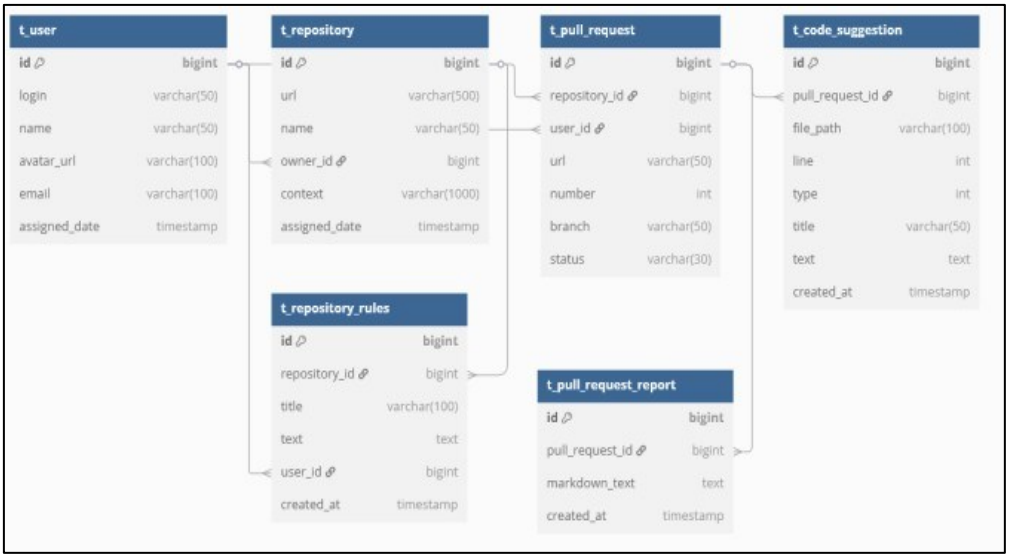

Figure 4. Data model applied to the prototype.

Figure 5 (Fragment of the Prompt Sent to the OpenAI API) presents the final structure of the prompt sent to the OpenAI API during the analysis process:

```
You are a code reviewer specialized in software engineering and best practices. Carefully analyze the following code
and identify possible *code smells*.

Your analysis should consider the following criteria:

---

🔎 **Code Smells to Identify**

During the analysis, identify and suggest corrections for any existing *code smells*, focusing especially with
**special attention and rigor** in identifying Feature Envy, Large Class / God Class and Long Method:

Feature Envy
Happens when a method in one class is overly interested in the data or methods of another class.
- Methods that access multiple fields or methods from another class more than their own;
- Patterns like: `external.getSomething().doAnotherThing()` or **heavy use of getters from other objects**;
- If the method under analysis repeatedly invokes methods from the same external instance (such as metrics.getX()
multiple times), this should be considered a **strong** indication of Feature Envy.
- Even if the class that owns the data is not included in the context, propose the creation of a new method in that
class to encapsulate the logic.
- Cases where logic could be **moved to the data-owning class** for better cohesion;
- Use of foreign data manipulation instead of internal responsibility handling.
Large Class / God Class
A class that takes on too many responsibilities and grows beyond its intended purpose.
- Becomes hard to read, test, and maintain;
- Often manipulates unrelated concerns like persistence, presentation, and domain logic;
- Breaks encapsulation and the SRP.

Long Method
This smell exists when a method tries to do too much, even if it's short in lines.
- Often mixes concerns like business logic, validation, and I/O;
- Increases complexity and reduces reusability;
- Difficult to read and test in isolation.

Divergent Change
This smell appears when a single class requires modifications for different reasons, such as new features, format
changes, or persistence adjustments.
- Indicates an accumulation of unrelated responsibilities in the same module;
- Violates the Single Responsibility Principle (SRP);
- Makes maintenance more difficult and error-prone.
Shotgun Surgery
This occurs when a single change requires edits in many different places across the codebase.
- Suggests poor encapsulation and scattered logic;
- Increases the chance of bugs and maintenance cost;
- Slows down development and complicates refactoring.

Long Parameter List
This smell is present when a method or function accepts an excessive number of parameters.
- Reduces readability and increases cognitive load;
- Often indicates that parameters should be grouped into a meaningful object;
- Makes method calls harder to understand and maintain.

Data Clumps
Occurs when certain variables tend to appear together across various places.
- Indicates those variables should likely be encapsulated in a class;
- Repeating groups reduce reusability and increase duplication;
- Introduces unnecessary complexity when spread out.
Message Chains
Detected when objects call methods that call other methods in succession (e.g., `a.getB().getC().doSomething()`).
- Leads to brittle code and high coupling;
- Any change in the chain's structure propagates across the codebase;
- Violates encapsulation and hides true dependencies.

Middle Man
A class that delegates most of its functionality to another class without adding real value.
- Increases indirection and reduces clarity;
- Clients should interact directly with the useful class;
- Suggests the intermediary class could be removed.

Speculative Generality
Appears when code is generalized for reuse that hasn't happened—and might never happen.
- Includes unused interfaces, abstract classes, or parameters "just in case";
- Adds unnecessary complexity and slows development;
- Should be removed until there's a real need.
```

Figure 5 – Final Prompt Fragment submitted to the OpenAI API

3.1. General Format, Page Layout and Margins

The experimental protocol was divided into stages. First, the files containing code smells were separated by category. They were then sent based on their category by pull requests to ensure traceability of the results. For each test case, it was recorded whether the tool detected the expected code smell and, when applicable, whether a refactoring action was also suggested.

Table 3 (Results of Code Smell Detection and Suggestion Tests) presents a quantitative summary of the tests performed for each type of code smell. The total number of evaluated files is displayed, how many of them were correctly identified by the tool as containing smells, and, within these, how many received correction suggestions from the model.

Table 3. Results of the Evaluation of the Detection and Suggestion of Code Smells.

Code Smell	**Evaluation**				
	Tested cases	Detections	Detection rate (%)	Suggest ed correction	Rate with correction
Long Method	10	10	100%	10	100%
God/Large Class	10	9	90%	8	80%
Feature Envy	10	10	100%	6	60%

As observed, the Long Method type showed maximum performance in both metrics, with 100% detection and 100% correction suggestions in the cases analysed. This result may be related to the nature of this smell, which often has structural features that are easily recognizable by language models trained to interpret large blocks of code.

In the case of God/Large Class, the tool detected 9 of the 10 cases evaluated, which represents a detection rate of 90%. Of these, 8 were accompanied by suggestions for correction, totalling a rate of 80%. The slight drop in the detection rate can be explained by the subjectivity involved in defining the threshold at which a class is considered excessively large or with multiple responsibilities. This subjectivity can also have an impact on the generation of specific suggestions, given the need for more refined semantic interpretation by the model.

Feature Envy had a detection rate of 100%, with 6 of the 10 cases receiving refactoring suggestions. Despite the complete identification of cases, the correction rate was lower (60%). This suggests that, although the model can recognize signs of excessive access to data from other classes — the main characteristic of smell — the generation of contextually appropriate suggestions can be more challenging, requiring a greater understanding of the logic of relationships between objects.

In general, the results demonstrate that the tool has high effectiveness in the task of identifying code smells, with detection rates equal to or greater than 90% for all types analysed. The rate of generation of correction suggestions was more variable, indicating that, although AI is effective in detection, there are still limitations in proposing specific improvements in more complex or ambiguous scenarios.

3. CONCLUSIONS

During the development of this work, it was found that, during the code review process, recurring problems related to poor code structuring may arise, known as code smells. These issues, if not identified and corrected, can compromise software quality and increase technical debt over time. To minimize these impacts and promote higher quality in development, it was proposed to create a prototype based on generative artificial intelligence, the prototype Code Reviewer, capable of automatically detecting code smells and suggesting improvements during the code review process.

In this sense, the central purpose of the research was to investigate the use of generative AI, through models such as GPT-4, to assist in the identification and correction of code smells in an automated way. Based on the theoretical review and practical analysis of the results obtained, it was possible to propose a concrete solution to this challenge.

It was sought to study the code review process, understanding its stages, challenges, and importance in the development cycle. Through an in-depth literature investigation, it was possible to recognize the limitations of manual approaches, which supported the need for automated tools such as the proposed prototype.

The research on code smells allowed us to identify that these failures are common in the daily lives of developers and often neglected, making clear the relevance of a tool that acts in their early detection.

The Code Reviewer prototype represented the last stage of this work. The system is designed to integrate platforms such as GitHub and GitLab, automatically analysing pull requests and issuing technical suggestions. It was concluded that the need identified by the research was duly demonstrated, since most participants indicated that they were in favour of a tool to automate the code review process through the detection and suggestion of correction of code smells.

During the study, it was also possible to identify limitations in traditional static analysis tools, such as SonarQude. Although widely used, such tools operate with fixed and generic rules, which restricts their ability to understand the context of the system and the developer's intent. In many cases, they are unable to identify opportunities for generalization between similar functions or offer concrete suggestions for improvement, limiting themselves to pointing out the problems detected [17]. This scenario further reinforces the value of AI-powered approaches, which bring an additional layer of interpretation and context to code analysis.

AI-based models, unlike traditional static analysis, can perform dynamic and contextualized analysis. For example, they can identify that a seemingly small class is concentrating several responsibilities — such as database access, business logic, and interface control — characterizing itself as a God Class, even without crossing rigid lines or complexity limits. Situations like this are difficult to capture by static rules but can be perceived by intelligent systems trained on large volumes of code [18].

However, these models can still generate false positives, suggest inappropriate refactoring, and require frequent adjustments and a robust database. Thus, the recommendation is to use them as support for human work, combining them with manual reviews and traditional tools to enhance the quality of the software [19].

Thus, the automated detection of code smells through artificial intelligence represents a hybrid approach, which does not replace the developer's critical eye but complements it, strengthening

software engineering practices and contributing to the construction of cleaner, more organized and sustainable systems [19].

As illustrated in Figure 6 (Acceptance of AI in detecting and suggesting correction of code smells during code review), more than 95% of respondents said they agreed (27.1%) or strongly agreed (68.8%) that AI can improve detection and suggest corrections for code smells in this context.

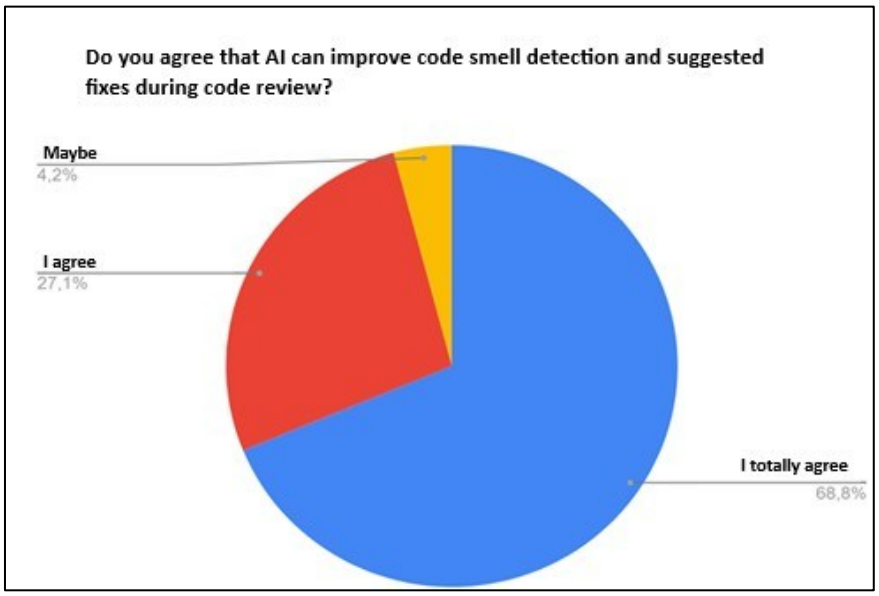

Figure 6. Acceptance of AI in detecting and suggesting correction of code smells during code review.

REFERENCES

[1] A. Singh, Agile & Scrum. Independently Published, 2019.

[2] K. Schwaber and J. Sutherland, "The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game," Billlewistraining.com, 2020.

[Online]. Available: https://billlewistraining.com/wpcontent/uploads/2017/02/PMP-Agile-Study-Materials.pdf. [Accessed: 29-Jul-2025].

[3] K. Beck, "Manifesto for Agile Software Development," Agilemanifesto.org, 2001. [Online]. Available: https://agilemanifesto.org. [Accessed: 29-Jul-2025].

[4] M. Fowler, Refactoring: Improving the design of existing code, 2nd ed. Boston, MA: Addison Wesley, 2019.

[5] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," Empir. Softw. Eng., vol. 21, no. 5, pp. 2146–2189, 2016.

[6] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in 2013 35th International Conference on Software Engineering (ICSE), 2013.

[7] R. C. Martin, "The clean coder: a code of conduct for professional programmers," Ebsco.com, 2011. [Online]. Available:

https://research.ebsco.com/linkprocessor/plink?id=938c30ca-6323390e-b53d-a369a69ca311. [Accessed: 29-Jul-2025].

[8] T. Hombergs and T. Hombergs, “Get Your Hands Dirty on Clean Architecture,” Ebsco.com, 2019. [Online]. Available:

https://research.ebsco.com/linkprocessor/plink?id=3d7eec8d-3697301b-9e1d-915e0c6a470e. [Accessed: 29-Jul-2025].

[9] M. Zhang, T. Hall, and N. Baddoo, “Code Bad Smells: a review of current knowledge,” J. Softw. Maint. Evol.: Res. Pract., vol. 23, no. 3, pp. 179–202, 2011.

[10] A. S. Cairo, G. de F. Carneiro, and M. P. Monteiro, “The impact of code smells on software bugs: A Systematic Literature Review,” Information (Basel), vol. 9, no. 11, p. 273, 2018.

[11] M. Soni, “Defect prevention: Reducing costs and enhancing quality,” isixsigma.com, 15-Oct-2024. [Online]. Available:

https://www.isixsigma.com/software/defect-prevention-reducingcosts-and-enhancing-quality/. [Accessed: 29-Jul-2025].

[12] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Nashville, TN: John Wiley & Sons, 2008.

[13] J. Martins, C. Bezerra, A. Uchôa, and A. Garcia, “How do code smell co-occurrences removal impact internal quality attributes? A developers’ perspective,” in Brazilian Symposium on Software Engineering, 2021.

[14] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” J. Syst. Softw., vol. 167, no. 110610, p. 110610, 2020.

[15] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, “Code smells detection and visualization: A systematic literature review,” Arch. Comput. Methods Eng., vol. 29, no. 1, pp. 47– 94, 2022.

[16] S. Tandon, V. Kumar, and V. B. Singh, “Study of code smells: A review and research agenda,” Int. j. math. eng. manag. sci., vol. 9, no. 3, pp. 472–498, 2024.

[17] Y. Zhang, C. Ge, H. Liu, and K. Zheng, “Code smell detection based on supervised learning models: A survey,” Neurocomputing, vol. 565, no. 127014, p. 127014, 2024.

[18] T. Lewowski and L. Madeyski, “Code smells detection using artificial intelligence techniques: A business-driven systematic review,” in Developments in Information & Knowledge Management for Business Applications, Cham: Springer International Publishing, 2022, pp. 285– 319.

[19] U. Cihan et al., “Automated Code Review In Practice,” arXiv [cs.SE], 2024.