AI-Assisted Code Review and Quality Assurance: A Comparative Analysis of Code Smell and Security Vulnerability Detection

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
June 2026
Yasmeen Saher

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Yasmeen Saher: AI-Assisted Code Review and Quality Assurance: A Comparative Analysis of Code Smell and Security Vulnerability Detection

Master of Science (Tech) Thesis, 60 p., 12 app. p.
Software Engineering
June 2026

Code review is a fundamental practice in software engineering for ensuring code quality, maintainability, and security, yet it remains largely manual, time-consuming, and dependent on individual expertise. As software systems increase in size and complexity, traditional review processes and rule-based static analysis tools struggle to maintain consistency and accuracy, often failing to detect deeper design flaws, contextual defects, and complex security vulnerabilities. Advances in artificial intelligence (AI) and machine learning (ML), particularly in natural language processing and deep learning, have introduced new opportunities to enhance code review automation by enabling models to learn from large codebases and historical review data to identify code smells, defects, and security issues beyond the capabilities of traditional approaches. Despite growing interest in these AI-assisted techniques, their effectiveness, reliability, and practical integration into real-world development workflows remain insufficiently explored.
This thesis investigates AI-assisted code review tools through a two-part approach. First, a literature review of 26 empirical studies is conducted to identify current research trends, evaluated techniques, and common limitations in AI-based code smell and vulnerability detection. Second, an experimental study is performed using selected Large Language Models (Codex 5.5, GPT-5.5, and Claude Sonnet 4.6) and a static analysis tool (SonarQube) to compare their performance on code smell and security vulnerability detection tasks. To the best of our knowledge, this is the first study to evaluate Codex 5.5 for code smell detection, and one of the first to jointly investigate both code smell and security vulnerability detection within a unified experimental framework. The results indicate that LLM-based approaches outperform the static analysis tool in terms of accuracy, recall, and F1-score across both code smell and vulnerability detection tasks. However, the study also identifies limitations such as prompt dependency, context window constraints, and non-deterministic model behavior, which may affect reproducibility and consistency of results.

Keywords: Code smells, AI-assisted code review, software quality assurance, software defects, security vulnerabilities, large language models, static analysis

Contents

1 **Introduction** 1

2 **Background** 6

2.1 Code Smells . . . 6

2.2 Security Vulnerabilities . . . 7

2.3 Artificial Intelligence (AI) . . . 9

2.4 Machine Learning (ML) . . . 9

2.5 Deep Learning (DL) . . . 10

2.6 Large Language Models (LLMs) . . . 11

2.7 Performance Metrics . . . 12

2.7.1 Accuracy . . . 12

2.7.2 Precision . . . 12

2.7.3 Recall . . . 12

2.7.4 F1-Score . . . 13

3 **Code Review** 14

3.1 Purpose and Importance . . . 14

3.2 Traditional vs. Automated Code Review . . . 15

3.3 Evolution of AI in Code Analysis . . . 16

3.4 Large Language Models (LLMs) for Code Review . . . 19

3.5 Existing AI-Assisted Code Review Tools . . . 21

3.6 Reported Benefits and Challenges 22
3.7 Research Gaps Identified in Literature 24

4 Research Methodology **26**

4.1 Research Approach 26
4.1.1 Data Collection and Selection Criteria 27
4.1.2 Study Screening 28
4.1.3 Data Extraction and Evaluation Framework 28

5 Comparative Analysis of Existing Studies **30**

5.1 Summary of Key Findings from Prior Studies 30
5.2 Performance Comparison Across Models 31
5.3 Common Issues Detected by AI Tools 32
5.4 Accuracy and Precision 33
5.5 Limitations and Developer Productivity 34

6 Experiment and Results **36**

6.1 Dataset Selection 36
6.2 Experimental Setup 37
6.2.1 Evaluation Metrics and Confusion Matrix 38
6.2.2 Code Smell Detection 39
6.2.3 Vulnerability Detection 43

7 Discussion **48**

7.1 Key Findings 48
7.2 Comparison with Literature Findings 50
7.2.1 Code Smells 50
7.2.2 Vulnerability Detection 51
7.3 Limitations and Threat to Validity 53

8 Conclusion **55**
8.1 Summary 55
8.2 Answers to Research Questions 57
8.3 Future Work 59

References **1**

Appendices

A **1**

B **1**

List of Figures

3.1 AI Evolution. 17
3.2 Comparison of traditional ML and deep learning approaches. 18

4.1 Research Process Stages. 28

List of Tables

2.1 Code Smells Definitions . . . 7

3.1 Comparison of Detection Approaches Based on Existing Studies . . . 20

6.1 Confusion matrix for evaluation . . . 39
6.2 Distribution of code smell samples . . . 40
6.3 Code Smell - Overall Confusion Matrix Results . . . 41
6.4 Confusion Matrix Results by Code Smell Type . . . 41
6.5 Code Smell - Overall Performance Metrics . . . 42
6.6 Performance Metrics by Code Smell Type . . . 42
6.7 Severity detection accuracy on true positive samples . . . 43
6.8 List of CWEs used in the experiment . . . 44
6.9 Security Vulnerability - Confusion Matrix Results . . . 46
6.10 Security Vulnerability - Performance Metrics . . . 46

A.1 Data Extraction of Previous Studies . . . 1

1 Introduction

Software quality assurance is a critical aspect of modern software engineering, directly influencing system reliability, maintainability, and security. Among quality assurance practices, code review plays a central role by enabling early detection of defects, enforcing coding standards, and facilitating knowledge sharing among developers. Traditionally, code review has been conducted manually through peer review processes, where developers inspect source code to identify defects, code smells, and potential security vulnerabilities. While effective, manual code review is inherently time-consuming, subjective, and heavily dependent on the experience and availability of reviewers [1], [2], [3]. As software systems continue to grow in size, complexity, and development velocity, maintaining consistent and thorough manual reviews has become increasingly challenging [1], [2].

To address scalability concerns, static analysis tools have been widely adopted to automate parts of the code review process. These tools rely on predefined rules, heuristics, and code metrics to detect syntactic errors, violations of coding standards, and known vulnerability patterns [4]. However, traditional static analysis approaches are limited in their ability to capture deeper design flaws, contextual defects, and semantic relationships within code. As a result, they often fail to detect complex code smells or subtle security issues, reducing developer trust and limiting their practical effectiveness [5].

Advances in artificial intelligence (AI) and machine learning (ML) have intro-

duced new opportunities to enhance automated code analysis and review. Unlike rule-based systems, machine learning models can learn patterns from historical code repositories, labeled datasets, and past review outcomes [6]. This enables AI-assisted code review tools to identify code smells, defects, and vulnerabilities that may not be explicitly encoded as rules. Techniques from natural language processing (NLP) and deep learning further allow models to treat source code as structured or semi-structured language, capturing syntactic, semantic, and contextual information [7].

Code smell detection using machine learning has emerged as a particularly active research area within AI-assisted software engineering. Several studies have demonstrated that supervised machine learning techniques can effectively identify multiple types of code smells when trained on software metrics and structural features. For example, Mahalakshmi et al. reported high detection accuracy across multiple machine learning models, indicating that ML-based approaches can significantly outperform traditional rule-based detectors under controlled conditions [8].

Furthermore, machine learning can enhance static analysis tools by automatically detecting vulnerabilities in web applications, as demonstrated by the DEKANT tool finding 16 zero-day vulnerabilities in open-source PHP applications and WordPress plugins [9]. Some modern static analysis tools incorporate AI or ML components, making them hybrid systems rather than purely rule-based tools such as SonarQube[1] (advanced editions) and DeepCode (Synk Code)[2].

Despite these promising results, existing research also highlights important limitations. Di Nucci et al. showed that machine learning models trained on datasets containing multiple code smell types often suffer from reduced performance compared to models trained on single-smell datasets, suggesting challenges related to feature generalization and class imbalance [10]. Furthermore, the literature indicates that traditional machine learning models based on handcrafted structural metrics,

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

[2]https://www.snyk.io/

such as size, complexity, coupling, and cohesion, frequently achieve strong and stable performance across datasets. While deep learning techniques offer potential advantages in capturing semantic information, their effectiveness remains highly dependent on dataset size, labeling quality, and model interpretability. As a result, no single AI-based approach has yet emerged as a definitive replacement for traditional code review practices.

Another significant challenge in evaluating AI-assisted code review tools is the lack of standardized datasets and evaluation methodologies. Studies often rely on heterogeneous datasets, different labeling strategies, and varying performance metrics, making direct comparison difficult. Reported results vary widely, and inconsistencies in experimental design limit the generalizability of findings. Moreover, practical concerns such as explainability, developer trust, and integration into existing development workflows remain underexplored, despite their importance for real-world adoption.

In addition to the challenges identified in the literature, there remains few studies on the effectiveness of the latest generation of large language models for software quality assessment. While previous studies have evaluated models such as GPT-3.5, GPT-4, Claude 3, and various machine learning approaches for either code smell detection or vulnerability detection, no studies have examined both tasks within a unified evaluation framework. To the best of our knowledge, this is the first study to evaluate Codex 5.5 for code smell detection and one of the first studies to jointly investigate both code smell and security vulnerability detection using LLMs and a traditional static analysis tool. By combining a systematic literature review with an exploratory experimental evaluation, this thesis provides updated evidence on the capabilities and limitations of AI-assisted code review tools and compares their effectiveness with traditional static analysis tools. This thesis addresses the following research questions:

RQ1: How do AI-assisted code review tools perform compared to traditional static analysis in terms of defect detection accuracy and efficiency?

RQ2: What types of software quality issues, such as code smells and security vulnerabilities, are most commonly detected by AI-assisted code review tools compared to traditional approaches?

RQ3: What challenges and limitations are reported in existing studies regarding the reliability, interpretability, and practical adoption of AI-assisted code review tools?

The remainder of this thesis is organized as follows. Section 2 presents the background concepts relevant to this study. Section 3 reviews the code review process, discussing the evolution of AI in code analysis, existing AI-assisted code review tools, and identifying research gaps in the literature. Section 4 describes the research methodology, including the research approach, study selection process, and data extraction framework. Section 5 presents a comparative analysis of existing studies on AI-assisted code review tools, focusing on their performance, commonly detected issues, limitations, and impact on developer productivity. Section 6 describes the experimental design, dataset selection, evaluation metrics, and the results of the code smell and vulnerability detection experiments. Section 7 discusses the findings, compares the experimental results with previous literature, and outlines the limitations and threats to validity of the study. Finally, Section 8 concludes the thesis by summarizing the key findings, answering the research questions, and presenting directions for future work.

Artificial Intelligence tools were used in this thesis to support specific academic tasks. ChatGPT was used to improve the clarity and readability of the text, reduce redundancy, and assist in formatting tables for the thesis. All AI-assisted outputs were carefully reviewed, revised, and validated by the author before inclusion in the final document. In addition, Gemini was used as a supportive tool for locating

and extracting relevant information from selected literature sources in Section 5. The extracted information was cross-checked against the original publications to ensure accuracy before being incorporated into the thesis. Large Language Models were also utilized as part of the experimental study to evaluate their performance in code smell and vulnerability detection. The design, execution, and analysis of the experiments were fully conducted and validated by the author.

2 Background

2.1 Code Smells

Code smells are surface indicators of deeper design problems in source code; while not necessarily bugs, they reflect structural weaknesses that affect maintainability, readability, and extensibility [11] [12]. Although code smells do not directly cause software failures, they increase technical debt and make software systems more difficult to understand, modify, test, and evolve over time [11], [13]. The previous study highlights that the presence of code smells is associated with higher maintenance effort, increased fault proneness, and reduced developer productivity, highlighting the importance of their early detection and refactoring [13].

Code smells are generally categorized into two main levels based on their scope of occurrence within the software system: class-level and method-level smells. Class-level smells refer to design issues that affect the overall structure of a class, such as Blob (God Class) and Data Class, where responsibilities are improperly distributed across classes. In contrast, method-level smells occur within individual methods and typically reflect poor implementation or excessive complexity, such as Feature Envy and Long Method. As noted by developers in the previous study, excessive complexity and size mean they have to spend "more time for understanding than for coding" just to remember what a method was about after reading it [13]. The distinction between code smell types is important, as different smells require different

detection strategies and reflect different levels of design degradation. The definitions of the code smells discussed in this study are presented in Table 2.1.

Code Smells	Definitions
Data Class	A class that contains only fields and simple accessor methods such as getters and setters, with little or no behaviour of its own [11], [14].
God Class	A class that centralises excessive responsibilities, leading to high complexity and low cohesion [14], [15], [16]. This smell is also referred to as Blob.
Feature Envy	A method that relies more on the attributes of another class than on its own, indicating misplaced responsibilities [11], [14].
Long Method	A method that contains too many operations, reducing readability, maintainability, and clarity [11], [14], [15], [16].

Table 2.1: Code Smells Definitions

2.2 Security Vulnerabilities

Security vulnerabilities are weaknesses or flaws in software design, implementation, or configuration that can be exploited by attackers to compromise the confidentiality, integrity, or availability of a system[1]. Unlike code smells, which primarily affect maintainability, security vulnerabilities can lead to unauthorized access, data breaches, privilege escalation, denial-of-service attacks, and other security incidents.

To systematically study and evaluate security vulnerabilities, researchers rely on the Common Weakness Enumeration (CWE); a community-developed taxonomy that categorizes software security flaws into distinct, measurable profiles. When evaluating object-oriented applications, such as systems written in C, java, python, vulnerabilities are analyzed across prominent operational categories. Key CWE categories commonly evaluated using standardized frameworks like the NIST Juliet Test Suite include:

[1] https://csrc.nist.gov/glossary/term/vulnerability

- Injection Flaws (e.g., CWE-89: SQL Injection, CWE-78: OS Command Injection): These occur when untrusted user input is passed directly to an interpreter (such as a database engine or system shell) without proper validation or escaping[2]. The interpreter executes the unintended malicious data as part of a command, resulting in data exfiltration or arbitrary code execution.

- Cross-Site Scripting (CWE-79: XSS): Prevalent in web applications and services, XSS flaws manifest when an application includes untrusted data in a web page without proper validation or encoding. This allows attackers to execute malicious scripts within a victim's browser session[3].

- Improper Resource Management (e.g., CWE-400: Uncontrolled Resource Consumption, CWE-764: Multiple Locks): These flaws occur when a system fails to properly allocate, track, or release critical resources (such as memory, file handles, or thread locks)[4] [5]. This can easily lead to a Denial of Service (DoS) state or deadlocks within multithreaded applications.

- Broken Authentication and Cryptographic Flaws (e.g., CWE-327: Use of a Broken or Risky Cryptographic Algorithm, CWE-259: Use of Hard-coded Password): These weaknesses involve the structural mishandling of sensitive data protection, such as relying on weak hashing algorithms or baking credentials directly into the source code architecture[6].

Code smells and security vulnerabilities can be detected using traditional static analysis, dynamic analysis and machine learning techniques. Large Language Models (LLMs) further extend these capabilities by incorporating semantic and contextual

[2]https://owasp.org/Top10/2025/A05_2025Injection/
[3]https://cwe.mitre.org/data/definitions/79.html
[4]https://cwe.mitre.org/data/definitions/764.html
[5]https://cwe.mitre.org/data/definitions/400.html
[6]https://owasp.org/Top10/2025/A04_2025Cryptographic_Failures/

understanding of source code, enabling the detection of vulnerabilities that may not be captured by rule-based approaches alone [17].

2.3 Artificial Intelligence (AI)

Techniques that enable computers to mimic human decision-making [18]. It refers to the capability of computational systems to perform tasks that would typically require human intelligence, including reasoning, learning, problem-solving, perception, and language understanding. Artificial intelligence systems achieve this by utilising data-driven approaches and computational models that allow them to identify patterns, make predictions, and improve performance over time without explicit rule-based programming for every possible scenario [19].

At its core, AI can be divided into two broad categories: *narrow AI*, which refers to systems designed to perform specific tasks such as image recognition, speech processing, or code analysis, and *general AI*, which refers to hypothetical systems capable of performing any intellectual task that a human can [20]. Contemporary AI systems used in software engineering, including those evaluated in this study, fall within the category of narrow AI, as they are designed and optimised for specific tasks such as code review, vulnerability detection, and code smell identification.

2.4 Machine Learning (ML)

Machine learning (ML) is a subfield of artificial intelligence (AI) that enables computational systems to learn patterns and make decisions from data without being explicitly programmed for each specific task [18]. Instead of relying on rigid, hard-coded rule sets engineered by human experts, ML frameworks utilize mathematical, optimization, and statistical techniques to analyze input data and iteratively minimize a loss function to improve predictive performance over time [21].

Machine learning approaches are broadly categorised into three learning paradigms based on the nature of the training signal available [22]. *Supervised learning* involves training a model on labelled data, where each input is paired with a corresponding output label, enabling the model to learn a mapping function that generalises to unseen examples. Common supervised learning algorithms include decision trees, random forests, support vector machines, and neural networks. *Unsupervised learning* involves training on unlabelled data, where the model identifies inherent structure, patterns, or clusters without explicit guidance [22]. *Reinforcement learning* involves an agent that learns by interacting with an environment, receiving rewards or penalties based on its actions, and optimising its behaviour to maximise cumulative reward over time [22].

2.5 Deep Learning (DL)

Within supervised learning of ML, *deep learning* has emerged as a particularly powerful paradigmAs software systems grow in scale and architectural complexity, traditional ML models are often limited by the need for manual feature engineering. To mitigate this constraint, Deep Learning utilizes multi-layered artificial neural networks to autonomously extract high-level hierarchical representations from raw code structures [18], [19]. Deep learning models have demonstrated remarkable performance across a wide range of tasks including image recognition, natural language processing, and code analysis.

In modern code analysis frameworks, DL architectures including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Transformer-based models (e.g., CodeBERT) do not merely look at simple software metrics. Instead, they process source code by converting Abstract Syntax Trees (ASTs) or Control Flow Graphs (CFGs) into dense vector embeddings [23]. This allows the narrow AI pipeline to capture semantic, syntactic, and structural dependencies across

code blocks, significantly improving the overall accuracy and viability of automated bug, code smell, and vulnerability detection engines.

2.6 Large Language Models (LLMs)

In recent years, the paradigm of contemporary AI-assisted software analytics has been advanced by the introduction of Large Language Models (LLMs). LLMs are deeply layered neural networks typically containing billions of parameters trained on massive corpora of natural language text and source code repositories [24]. While early Machine Learning models required manual feature engineering and early Deep Learning models relied on specialized graph structures, LLMs handle code text sequentially, treating software source code as a highly structured language governed by rigid contextual grammars [25].

The capability of LLMs to analyze complex software systems stems from the Transformer architecture [26]. Unlike historical Recurrent Neural Networks (RNNs) that process source code tokens strictly in chronological order, Transformers utilize self-attention mechanisms. This mathematical framework allows the model to calculate the contextual relationship between code elements regardless of their distance from one another within a file. In code smell or vulnerability detection, this capability is vital. For instance, a method invoking a sensitive variable or an un-encapsulated class reference dozens of lines away creates an intricate data-dependency line. The attention layers within an LLM dynamically weigh these distant connections, creating a comprehensive semantic map of the code block.

2.7 Performance Metrics

2.7.1 Accuracy

Accuracy is the proportion of total predictions (both positive and negative) that were correct. It is the most intuitive metric but can be misleading in "imbalanced" datasets where one class significantly outnumbers the other [27].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

TP: True Positives, TN: True Negatives, FP: False Positives, FN: False Negatives

2.7.2 Precision

Precision measures the "exactness" of a classifier. It answers the question: Of all instances predicted as positive, how many were actually positive? High precision indicates a low false-positive rate [27].

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

2.7.3 Recall

Recall measures the "completeness" of a classifier. It answers the question: Of all actual positive instances, how many did the model correctly identify? High recall indicates a low false-negative rate [27].

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

2.7.4 F1-Score

The F1-score is the harmonic mean of precision and recall, providing a balanced measure when there is an uneven class distribution [27].

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.4}$$

3 Code Review

3.1 Purpose and Importance

Code review is a software quality assurance practice involving the assessment of code written by teammates to increase code quality [28]. It is one of the most widely practised quality assurance techniques in software development, applied across both industrial and open-source projects to detect bugs before they propagate into production systems [3], [29]. By enabling the early detection of coding issues, code review improves software reliability, supports maintainability, and reduces the cost of fixing defects in later stages of the software development lifecycle.

Code review can be conducted using various approaches, including pair programming, informal peer reviews, and formal inspections [3], each offering different levels of rigor and effectiveness depending on project context. When properly implemented, this accelerates and streamlines the software development process more effectively than many other quality assurance practices [3]. In this contemporary world, Modern Code Review (MCR), which is commonly integrated into collaborative version control platforms such as GitHub and GitLab, is widely recognized by software practitioners as an essential practice directly associated with improved product quality [30]. Beyond defect detection, code review plays an important role in knowledge sharing, team collaboration, and skill development [29]. The overall effectiveness of the code review process, however, is highly dependent on the

availability of appropriate tools, well-defined processes, and contextual factors that support both reviewers and code authors throughout the different stages of review [31].

3.2 Traditional vs. Automated Code Review

Automated code review has gained increasing attention as a means to address the scalability and efficiency limitations of traditional manual code review. Unlike manual reviews, which rely heavily on reviewer expertise and available time, automated tools can analyze large codebases continuously and consistently. Empirical studies indicate that such tools can provide tangible benefits when integrated into development workflows. For example, Kim et al. reported that approximately 60% of the issues identified by their automated Code Review Bot were resolved before product release, and more than 70% of participating developers expressed positive perceptions regarding its usefulness [32]. Similarly, Gunawan et al. observed that automated analysis tools contributed to reductions in defect density ranging from 15 to 30% and increased pull request throughput by up to 40%, with tools like Copilot Autofix[1] dramatically reducing vulnerability remediation times from 180 to 22 minutes for XSS issues [33].

Despite these advantages, existing evidence also highlights important limitations that prevent automated code review from fully replacing traditional human-led reviews. Kathiresan et al. emphasize that while Large Language Models (LLMs) introduce powerful semantic reasoning capabilities, their predictions are not consistently reliable and may suffer from false positives, hallucinated explanations, and high computational costs [34]. Furthermore, Tufano et al. found that ChatGPT generated code reviews are largely accepted by developers (with 89% of issues adopted),

[1]https://github.blog/news-insights/product-news/secure-code-more-than-three-times-faster-with-copilot-autofix/

they tend to narrow reviewer focus toward highlighted code regions, surface more trivial rather than severe issues, and fail to improve review efficiency or confidence due to the need for manual verification, thereby limiting their overall effectiveness as automated co-reviewers [35].

Overall, the literature suggests an emerging consensus that automated code review tools function best as a complementary mechanism rather than a standalone solution. While automation improves coverage and early defect detection, human reviewers remain essential for understanding contextual design decisions, validating complex logic, and making informed architectural judgments. Based on the reviewed literature, it seems that hybrid human–AI code review approaches are currently regarded as the most effective and practical strategy in modern software development.

3.3 Evolution of AI in Code Analysis

The analysis of source code has traditionally relied on static analysis techniques based on predefined rules and heuristics [36]. These rules are manually defined conditions that describe what constitutes a defect or code smell, while heuristics capture experience-based guidelines for identifying potentially problematic code structures. Garg et al. highlight that static analysis rules are traditionally created manually by domain experts [4]. Although these tools played an important role in automating repetitive and error-prone tasks in software development, their reliance on fixed rules limited their ability to capture complex design flaws, contextual issues, and deeper semantic relationships within code, often resulting in high false-positive rates [17].

To overcome the limitations of rule-based static analysis, researchers started using machine learning (ML) techniques for code analysis. Instead of relying on fixed rules, these approaches learn from existing code examples. Important characteristics (called features) are first extracted from the source code, such as code size, complexity, how classes depend on each other (coupling), and how well related the

methods inside a class are (cohesion) [10].

Traditional machine learning algorithms, such as decision trees, support vector machines, and random forests, are then trained using these features to decide whether a piece of code contains a code smell or not, as illustrated in Figure 3.2. Di Nucci et al. demonstrated that ML-based models could outperform traditional static analysis tools in detecting certain types of code smells (God Class, Feature Envy, Long Method, and Data Class), although their performance was highly dependent on feature quality and dataset balance [10].

Over the past decade especially, ML methods like classification models, neural networks, and embedding techniques have been increasingly used for source code analysis tasks previously handled only by rule-based methods. Research surveys report a consistent increase in the use of machine learning techniques for source code analysis since the early 2010s, indicating a growing shift toward data-driven approaches in software engineering tasks [37].

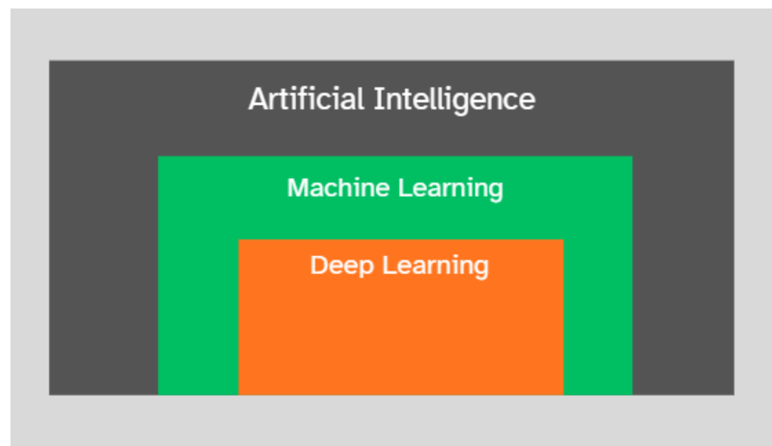

Figure 3.1: AI Evolution.

Figure 3.1 highlights that DL is a specialized branch of ML, which itself is a subfield of AI. The emergence of AI systems trained on large codebases accelerated evolution. By the mid-2020s, ML techniques have begun transforming code review practices. Instead of only flagging syntax or style errors, modern AI tools use

deep learning and natural language processing to understand context, detect more complex bugs, and suggest fixes. These tools can handle multiple programming languages and provide context-aware feedback, a significant departure from earlier static analysis, which lacked deep semantic understanding of code. Models such as OpenAI's Codex[2] use deep learning to translate natural language into code, provide intelligent completions, and assist developers interactively within IDEs. These tools represent a shift where code analysis and generation are informed by pattern learning from massive code repositories, not only by explicit rules.

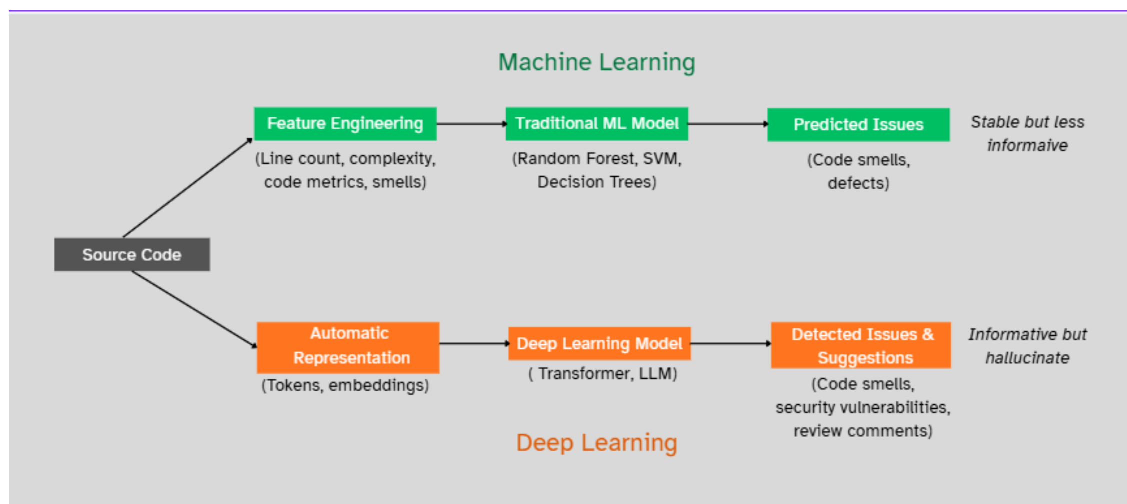

Figure 3.2: Comparison of traditional ML and deep learning approaches.

Figure 3.2 presents two approaches to applying AI in source code analysis. The traditional approach is generally stable and interpretable, but it is limited by predefined features. In contrast, deep learning models automatically learn representations from raw source code using embeddings and Transformer-based architectures. These models provide more context-aware and informative outputs, including vulnerability detection and review suggestions, but may produce hallucinated or inaccurate results.

In parallel, ML techniques have also been applied to security-focused code anal-

[2]https://openai.com/codex/

ysis. Zaharia et al. demonstrated that machine learning-based pattern recognition techniques can assist developers in identifying security-related issues in code as illustrated in 3.2, highlighting the potential of AI-driven approaches in improving secure software development practices [38].

Overall, the evolution of AI in code analysis reflects a transition from rule-based and metric-driven methods to data-driven, learning-based, and context-aware approaches. While AI techniques have shown promising results in detecting code smells, defects, and security vulnerabilities, existing research also highlights ongoing challenges related to accuracy, explainability, and practical adoption, motivating further investigation into AI-assisted code review systems.

3.4 Large Language Models (LLMs) for Code Review

Large Language Models (LLMs) demonstrate growing potential for supporting automated code review by assisting in issue detection, code improvement suggestions, and overall software quality enhancement. Their ability to reason over source code as structured text enables them to identify code smells, potential bugs, and readability or maintainability concerns. However, existing evidence also highlights important limitations that restrict their standalone applicability in professional development settings.

Recent empirical studies indicate that LLMs achieve moderate performance levels in automated code review tasks. For example, LLM-based systems correctly classify code correctness in approximately 63–68% of evaluated cases, while successful automated code corrections range between 54–68%, depending on task complexity and prompt design [39]. Other studies suggest that LLMs perform particularly well in identifying code smells, stylistic issues, and optimization opportunities, where

Table 3.1: Comparison of Detection Approaches Based on Existing Studies

Approach	**Description**	**Advantages**	**Limitations**
Manual Code Review	Human inspection of source code by developers or peers. *Examples:* Peer review, Gerrit, GitHub Pull Requests	High contextual and architectural understanding. Detects subtle design and semantic issues.	Time-consuming and costly. Poor scalability for large codebases.
Static Analysis Tools	Automated tools based on predefined rules and heuristics. *Examples:* SonarQube, PMD, CodeQL, SpotBugs	Fast and scalable. Consistent results.	High false positives. Limited to predefined rules.
Traditional Machine Learning	Supervised models using handcrafted metrics. *Examples:* Random Forest, SVM, Decision Trees	Higher accuracy than rule-based tools. Good performance with limited data. More interpretable models	Sensitive to dataset quality. Limited semantic understanding.
Deep Learning / LLMs	Neural models treating code as structured language. *Examples:* ChatGPT, Claude, DeepSeek	Captures semantic and contextual patterns. Minimal feature engineering. Detects complex smells	Data and compute intensive. Lower interpretability. Performance varies across datasets.

semantic understanding plays a larger role than strict syntactic precision [40].

Despite these promising results, several challenges remain. For instance, Kathiresan et al. emphasize that LLM-based code review systems can suffer from false positives, inconsistent accuracy, and hallucinated explanations, especially when applied to complex or unfamiliar codebases [34]. In addition, the computational cost and latency associated with large-scale LLM deployment raise concerns regarding practical integration into continuous integration pipelines.

Consequently, multiple researchers advocate for a human-in-the-loop approach, where LLMs serve as decision-support tools rather than autonomous reviewers [39].

This hybrid model allows developers to benefit from automated insights while retaining human judgment for validating recommendations and handling contextual or architectural decisions. Supporting this perspective, Goldman et al. observe that LLM-generated comments related to readability, bug localization, and maintainability are more likely to be acted upon by developers than purely stylistic suggestions, indicating tangible but selective practical value [41]. Table 3.1 summarizes the detection approaches used in code review, comparing their strengths and limitations based on findings from existing studies.

3.5 Existing AI-Assisted Code Review Tools

AI-assisted code review tools represent a rapidly evolving area within software quality assurance, leveraging machine learning techniques and Large Language Models (LLMs) to support developers in detecting defects, suggesting improvements, and improving maintainability. While AI-assisted review tools can reduce reviewer workload by providing automated feedback, they are positioned as complements to human expertise rather than replacements, as human reviewers remain essential for contextual judgment, trust decisions, and nuanced evaluation of code changes [42].

Several recent studies report encouraging results regarding the effectiveness of AI-assisted approaches, although performance varies considerably across tools and tasks. Almeida et al. introduced an IntelliJ IDEA plugin based on GPT-3.5 that analyzes Java code snippets and reported an issue detection accuracy of approximately 92% under controlled experimental conditions [43]. Rasheed et al. demonstrated that LLM-based models trained on large-scale code repositories can identify potential risks, detect code smells, and provide optimization suggestions, particularly for readability and maintainability-related concerns [40].

However, comparative evaluations indicate substantial variability in tool performance. Yetistiren et al. assessed the correctness of code generated by multiple

AI-assisted tools, including ChatGPT[3], GitHub Copilot[4], and Amazon CodeWhisperer[5] [44]. Their results indicate varying levels of code quality among these tools. ChatGPT generated correct code 65.2% of the time, followed by GitHub Copilot at 46.3% and Amazon CodeWhisperer at 31.1%, highlighting current reliability limitations[44]. These findings suggest that while AI-assisted code review can reduce manual review effort and highlight potential issues early, it does not consistently guarantee correctness. Supporting this perspective, Ayyarrappan et al. emphasize that AI-based review tools can help streamline code quality management and reduce long-term maintenance costs, but only when used alongside established development practices and human oversight [45].

Further reinforcing this view, Chowdhury et al. conducted a comprehensive study combining literature review, case studies, and experimental observations, concluding that AI-powered code reviews based on LLMs contribute positively to code quality and security [46]. Nevertheless, the authors stress that such systems should be employed as complementary tools, with human expertise remaining essential for validating recommendations. AI-assisted review enable human reviewers to shift their focus toward more strategic and complex evaluation tasks, while still maintaining oversight at the code level.

3.6 Reported Benefits and Challenges

One of the primary advantages of AI-assisted code review tools is real-time embedded feedback. Modern AI-based reviewers can be integrated directly into development environments or version-control workflows, allowing developers to receive immediate suggestions related to coding style, best practices, and code defects. For instance,

[3] https://chatgpt.com/

[4] https://github.com/features/copilot

[5] https://workshops.aws/categories/CodeWhisperer

Ernst et al. demonstrate that Artificial Intelligence Driven Development Environments (AIDEs) can now seamlessly integrate into popular IDEs like Visual Studio Code [47]. Manushree Vijayvergiya et al. further validate this by presenting AutoCommenter, a system that automatically learns and enforces coding best practices directly within version control systems [48]. This immediacy shortens feedback loops and reduces the time between code creation and quality improvement.

Another significant advantage is the semantic understanding of code. Unlike traditional static analysis tools, LLM-based reviewers can reason about code semantics and developer intent. Fried et al. demonstrate that generative models such as InCoder are capable of understanding incomplete or evolving code contexts, enabling more meaningful assessments beyond syntax [49]. This semantic awareness allows AI reviewers to identify code smells, maintainability issues, and potential future risks that rule-based tools often miss.

AI-assisted code review also contributes to reduced human cost and improved scalability. Large software projects generate substantial volumes of code changes, making thorough manual review difficult and time-consuming. In addition, as third-party libraries have become a significant part of applications, manual reviewing third-party components or libraries is also challenging due to potential malicious or vulnerable code. AI tools can automatically handle repetitive and low-level review tasks, such as checking naming conventions, formatting inconsistencies, and common anti-patterns. This automation allows human reviewers to focus on high-level architectural decisions, business logic, and critical security concerns, ultimately improving overall review efficiency and consistency [49].

Despite these benefits, AI-assisted code review systems face notable limitations. The major challenge is the lack of explainability and transparency in AI-generated review feedback. While AI tools often provide recommendations or warnings, they frequently fail to clearly explain the reasoning behind these suggestions. This limi-

tation can reduce developer trust and hinder adoption, particularly in safety-critical or security-sensitive environments. Developers may accept or reject recommendations without fully understanding their implications, increasing the risk of incorrect fixes or overlooked vulnerabilities.

Finally, AI-assisted code review introduces verification and accountability concerns. Although AI tools can enhance productivity, their outputs are not guaranteed to be correct or secure. As demonstrated in prior research on generative code models [49], AI-generated suggestions may appear plausible while still being logically flawed or insecure. Consequently, developers must invest additional effort to verify AI recommendations, which partially offsets the productivity gains and reinforces the need for human oversight.

3.7 Research Gaps Identified in Literature

Despite significant progress in applying machine learning and deep learning techniques to code smell detection, the existing literature exhibits several important gaps. Multiple studies reveal a lack of standardized datasets and comparable results due to heterogeneous data sources, limiting the reliability and replicability of research findings [50] [51]. Over half of studies use custom-curated datasets that are used only once, while publicly available benchmark datasets contain numerous flaws, including subjective labeling, limited smell coverage, small dataset sizes, and class imbalance issues, and are sparsely utilized [51]. The absence of standardized datasets and evaluation methodologies in existing research limits direct comparison between AI-assisted and traditional code review approaches. This inconsistency motivates RQ1, which seeks to comparatively analyze reported empirical findings in order to establish a clearer understanding of how AI-assisted code review tools perform relative to traditional static analysis.

Current datasets and evaluations demonstrably focus on a narrow subset of code

smells, with only a few types receiving substantial research attention. Only four smells; Blob, Feature Envy, Long Method, and Data Class have received sustained research attention [52], and a recent systematic review confirms that these few dominate the literature, leaving other smells underexplored and challenging [51]. Dataset analyses further reveal that most benchmarks primarily support God Class, Long Method, and Feature Envy, while several smells remain entirely unsupported [53]. RQ2 addresses this limitation by systematically investigating the range of issue types detected by AI-assisted tools, enabling a more comprehensive assessment of their practical utility.

Recent empirical studies indicate that AI-assisted code review tools face persistent challenges related to reliability, interpretability, and adoption. Large-scale analyses show that while many automated comments are acted upon, tools frequently generate inaccurate or irrelevant suggestions and may increase review time due to the need for human verification [39] [54]. Research also highlights limited explainability and insufficient context awareness as key weaknesses [55] [42], which contribute to trust issues and cautious adoption in practice [56] [57]. Overall, these findings demonstrate that despite their potential, AI-assisted code review tools remain constrained by reliability variability and human oversight requirements, directly motivating RQ3.

4 Research Methodology

4.1 Research Approach

This study adopts a mixed-method research design, combining a systematic literature review with a small-scale empirical evaluation. The mixed approach is selected to provide both breadth and depth in investigating AI-assisted code review tools. The systematic review enables a comprehensive synthesis of existing empirical evidence, while the empirical evaluation provides contextual validation and practical insights. The review follows established guidelines for systematic literature reviews in software engineering as proposed by Kitchenham [58]. This SLR approach was chosen to ensure methodological transparency, replicability, and comprehensive coverage of relevant empirical studies.

Given the rapid growth of AI-assisted code review research, a systematic synthesis is necessary to consolidate fragmented findings related to performance evaluation, issue-type coverage, and adoption challenges. The review focuses on peer-reviewed journal articles, conference papers, and high-quality preprints that evaluate AI-assisted code review tools, machine learning-based defect detection systems, and LLM-based review assistants. The goal is not merely to summarize prior studies, but to extract comparable data.

4.1.1 Data Collection and Selection Criteria

The data collection process was conducted through a structured search of major academic digital libraries, including IEEE Xplore, Scopus and Science Direct. The search string was constructed following Kitchenham's structured keyword identification process to ensure comprehensive yet focused retrieval of relevant studies. Based on the research questions, two primary conceptual groups were identified. The first conceptual group captures AI-based techniques applied to software analysis, while the second group reflects software quality assessment approaches, including traditional static analysis and manual review as well as specific issue types such as code smells and security vulnerabilities.

Within each concept group, synonyms were combined using the Boolean operator OR to maximize coverage. The two major concept groups were then connected using AND to ensure that retrieved studies addressed both concepts. The final search string is given below:

```
("Machine Learning" OR "Deep Learning" OR "Large Language Model" OR
LLM) AND ("Code Review" OR "Static Analysis" OR "Manual Code Review"
OR "Defect Detection" OR "Code Smell Detection" OR "Security
Vulnerability")
```

The search was limited to the following inclusion criteria:

- Publications written in English.
- Published between 2020 and 2026 to capture recent developments.
- Peer-reviewed journal articles and conference papers.
- Available for free online or through the University of Turku Library.
- Studies related to field of Computer Science.

4.1.2 Study Screening

The initial search results of 939 studies from selected databases were exported to a reference management tool called Zotero, and 39 duplicate records were removed before further screening, as illustrated in Figure 4.1. In the next stage, the titles and abstracts of the remaining papers were reviewed to assess their relevance to the research objectives. Studies were excluded if they were not related to software security, code analysis, vulnerability detection, or AI-based techniques. After applying these criteria, 117 papers were retained for further evaluation.

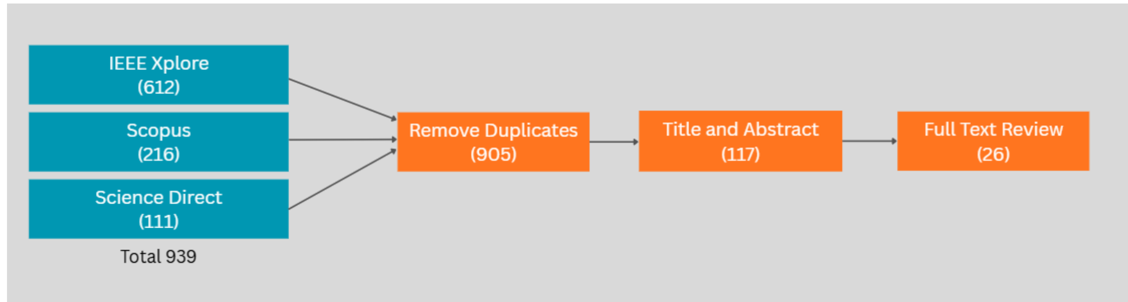

Figure 4.1: Research Process Stages.

Subsequently, the full text of the remaining studies was examined to determine their suitability for inclusion in the review. Papers were excluded if they were survey papers, focused only on datasets or tools without analysis, or lacked empirical evaluation. After the full-text assessment, 26 studies were selected as the final set of papers for detailed analysis in this thesis.

4.1.3 Data Extraction and Evaluation Framework

A structured data extraction table was developed through an iterative process based on the research objectives and established guidelines for systematic literature reviews. Initially, key data fields were identified by aligning the extraction requirements with the research questions (RQ1–RQ3) and commonly reported variables in related studies. These fields were then organized into a standardized template to

ensure consistency across all selected papers. To refine the form, a pilot extraction was conducted on a small subset of studies, allowing adjustments to be made for clarity, completeness, and relevance. The final version of the data extraction form was implemented using a tabular format, where each row represented a study and each column corresponded to a specific data attribute. The extracted data included the following elements:

- Type of AI technique used (ML, DL, LLM)
- Evaluation metrics (accuracy, precision, recall, F1-score, etc.)
- Types of issues detected (code smells, defects, vulnerabilities)
- Reported limitations and challenges

Following the data extraction process, a structured evaluation framework was applied to systematically analyze and compare the selected studies across three primary dimensions aligned with the research questions. The first dimension, performance evaluation, focuses on assessing how AI-assisted tools perform in comparison to traditional approaches, including static analysis, particularly in terms of detection accuracy and efficiency. The second dimension, issue-type coverage, examines the range and diversity of software quality issues, such as code smells and security vulnerabilities identified by different approaches. The third dimension, adoption and practical challenges, explores the reported limitations and barriers to real-world adoption, including concerns related to reliability, interpretability, scalability, and developer trust. This structured framework facilitates consistent cross-study comparison and supports the development of evidence-based conclusions regarding the effectiveness and limitations of AI-assisted code review systems, as illustrated in Table A.1 in Appendix A.

5 Comparative Analysis of Existing Studies

5.1 Summary of Key Findings from Prior Studies

The reviewed studies indicate that AI-assisted techniques, including machine learning, deep learning, and large language models, play a significant role in improving software quality analysis. Overall, these approaches demonstrate enhanced capability in detecting both code smells and security vulnerabilities compared to traditional rule-based static analysis, particularly in handling complex and context-dependent issues.

The findings show that AI-based methods are effective in identifying a wide range of defects, including design-related code smells and critical security vulnerabilities. However, their performance varies depending on the model type, dataset, and problem complexity. While some approaches achieve high accuracy and precision, challenges such as false positives, low recall in certain cases, and data-related limitations are frequently reported. In general, the studies suggest that hybrid approaches, combining AI techniques with traditional static analysis, provide more balanced and reliable results for practical software development environments.

5.2 Performance Comparison Across Models

The reviewed studies indicate that AI-assisted approaches, including machine learning, deep learning, and large language models, generally achieve higher detection performance compared to traditional static analysis techniques. Specifically, 20 studies ([59], [60], [61],[62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78]) explicitly report improvements in performance metrics such as accuracy, precision, recall, F1-score, or detection coverage when applying AI-based techniques.

Deep learning models show consistent performance gains in multiple studies. For instance, the BiGRU-based model proposed by Han et al. improves accuracy by 8.3% and F1-score by 17.7% [59], while the AutoReview framework presented by Yujia Chen reports an F1-score improvement of 18.7% [60]. Similarly, Steenhoek et al.'s DeepDFA framework achieves high precision and recall while maintaining efficient training performance through a graph-based hybrid learning approach [61]. In addition, Alawadi et al.'s federated learning-based FedCSD model achieves high accuracy (98.34%) while maintaining scalability [62].

Machine learning approaches also demonstrate strong performance, particularly when combining structural and semantic features. Zhang et al. and Ibba et al. report improved F1-scores and high accuracy (up to 0.977), highlighting the effectiveness of feature integration in vulnerability and code smell detection [63] [64].

Despite these improvements, traditional static analysis tools remain competitive in certain contexts. As reported by Santas Ciavatta et al., static analysis performs better in business-critical systems due to its reliability and consistency [65]. In contrast, LLM-based approaches show variable performance across studies. Sadik et al. indicate that while models such as GPT-4 achieve higher precision, they often suffer from low recall, particularly when analyzing complex or large-scale codebases [66].

Hybrid approaches combining AI techniques with static analysis demonstrate the most balanced performance. Studies by Chapman et al. and Feng et al. show that integrating LLMs with traditional analysis improves recall and F1-score, while mitigating limitations such as false positives and missed defects [67] [68]. Similarly, Santas Ciavatta et al. confirms that hybrid models outperform standalone AI or static approaches in non-critical scenarios [65].

Overall, the findings suggest that while AI-based models enhance detection performance, their effectiveness varies depending on the model type and application context. Hybrid approaches provide the most reliable and robust results across different software analysis tasks.

5.3 Common Issues Detected by AI Tools

The reviewed studies demonstrate that AI-assisted tools are capable of detecting a wide range of software quality issues, including both code smells and security vulnerabilities. For code smells, 11 studies (e.g., [62], [63], [66], [69], [70], [71], [72], [73], [74], [79], [80]) focus on detecting design and implementation issues such as God Class, Long Method, Feature Envy, Data Class, Brain Class, Blob, Message Chains, Refused Bequest, and other maintainability-related smells. These smells are associated with poor maintainability and increased defect risk. AI-based approaches improve detection by learning complex relationships between structural and semantic features. As shown by Ibba et al., integrating multiple feature types results in high detection accuracy up to 0.977 [64], while Steenhoek et al.'s graph-based and deep learning approaches further enhance the identification of complex smell patterns [61].

For security vulnerabilities, 14 studies ([17], [59], [60], [61], [64], [65], [67], [68], [75], [76], [77], [78], [81], [82]) demonstrate that AI-based techniques effectively detect a wide range of critical security issues, including SQL injection, command

injection, cross-site scripting (XSS), null pointer dereference, improper validation, information leak, buffer overflows, use-after-free errors and more. In addition, Zhang et al. highlights the ability of AI models to identify zero-day vulnerabilities by capturing semantic similarities between known and unknown patterns [63]. Security-related code issues, including hard-coded secrets, weak cryptographic practices, and empty passwords, are also detected using LLM-based semantic analysis, as reported by Pornprasit et al. and Zhang et al. [69], [83].

Overall, these findings indicate that AI-based approaches provide comprehensive coverage and improved capability in detecting both structural issues, such as code smells, and critical security vulnerabilities across different programming environments.

5.4 Accuracy and Precision

Table A.1 shows both the strengths and limitations of AI-assisted code review approaches in terms of accuracy, precision, and practical applicability. In terms of accuracy and precision, various studies (e.g., [59], [60], [62], [63], [64]) report high performance. For code smell detection, several studies reported strong classification performance across multiple smell categories. Khleel et al. [74] achieved very high detection accuracy using Bi-LSTM and GRU models, including 100% accuracy for Long Method detection. Similarly, Zhang et al. [69] used Graph Neural Networks (GNNs) for detecting Long Method and Blob smells, where most evaluation metrics exceeded 0.90. Ho et al. [71] further demonstrated that combining structural and semantic embeddings improves detection accuracy by up to 28.26%. In addition, federated learning approaches proposed by Ibba et al. [62] achieved high scalability with an accuracy of 98.34% for God Class detection. Other studies such as [63], [70], and [72] also reported improvements in precision, recall, and F1-score through the integration of semantic feature learning and static analysis techniques.

For security vulnerability detection, AI-assisted approaches also demonstrated strong performance across diverse vulnerability categories. Hybrid methods showed particularly notable results, for example, Ciavatta et al. [65] reported significant improvements over standalone approaches using multiple LLMs combined with Static Application Security Testing (SAST) tools, achieving approximately 0.90 accuracy and F1-score, while graph-learning and LLM-based approaches by Steenhoek et al. [61] achieved high precision (97.82%), recall (95.14%), and F1-score (96.46%). This trend was further supported by [60], [67], and [68], whose hybrid approaches combininLg LMs with static analysis improved recall and F1-score compared to standalone methods. Beyond hybrid techniques, deep learning approaches such as the BiGRU-based model proposed by Han et al. [59] improved accuracy by 8.3% and F1-score by 17.7%. In addition, the studies by [17], [77], and [78] reported that AI-based models outperformed conventional rule-based and baseline detection approaches in identifying software vulnerabilities.

5.5 Limitations and Developer Productivity

Several limitations of AI-assisted approaches were identified across the reviewed literature. One major challenge is the trade-off between precision and recall. For example, Sadik et al. [66] reported that GPT-4 achieved higher precision (0.79) than DeepSeek-V3 (0.42), but both models suffered from relatively low recall. Similarly, some machine learning models achieved high recall while generating a large number of false positives [76]. These issues can reduce developer trust and increase manual verification effort.

Another commonly reported limitation is hallucination and prompt dependency in LLM-based systems. Studies such as [82], [81], and [67] observed that LLMs can generate inconsistent or incorrect outputs, particularly when analyzing large or complex codebases. In addition, several studies highlighted scalability limita-

tions, context-window restrictions, and difficulties in handling multi-file projects. For example, Feng et al. [68] and Khare et al. [82] reported context limitations and reduced performance for large-scale software systems.

Data-related challenges were also frequently reported. Studies such as [71], [63], and [73] identified issues related to class imbalance, dataset dependency, and subjective labeling. These challenges affect model generalizability across different programming languages, project domains, and software architectures. Furthermore, deep learning and LLM-based approaches often require substantial computational resources, training cost, and parameter tuning, which limit their practical adoption in industrial environments.

Despite these limitations, AI-assisted approaches demonstrate meaningful improvements in developer productivity by automating defect detection, reducing manual inspection effort, and enabling earlier identification of issues in the development lifecycle. Pornprasit et al. demonstrated that fine-tuned LLMs improved exact match scores by 73-74% in automated code review tasks, directly reducing the review burden on developers [83]. Ciavatta et al. reported that combining LLMs with static analysis achieved an F1 of 0.90 across 11 security vulnerability types, reducing the manual effort required to identify security issues during development [65]. Furthermore, Yang et al. reported that LLM-generated static analysers discovered 92 new bugs beyond what traditional tools detected, demonstrating the potential to surface defects that would otherwise require extensive manual investigation [75]. Collectively, these findings suggest that while AI-assisted tools do not eliminate the need for human expertise, they serve as effective force multipliers that broaden detection coverage and reduce the repetitive cognitive demands of manual code review.

6 Experiment and Results

The experimental phase of this study was conducted between 10 May 2026 and 2 June 2026. This phase was exploratory in nature and served to complement and validate the findings of the literature review. It serves as an evaluation rather than providing statistically generalizable results to compare LLMs and traditional static analysis techniques on selected code smell and security vulnerability datasets. The objective is to provide empirical insights into the strengths and weaknesses of these approaches within the scope of this study.

6.1 Dataset Selection

To evaluate the effectiveness of AI-assisted code review techniques in detecting software quality issues, this study uses two complementary datasets that represent different categories of problems commonly found in source code. Software quality issues generally fall into two broad groups; code smells, which indicate poor design or maintainability concerns, and security vulnerabilities, which can be exploited by attackers.

- Code Smell Dataset: This study focuses on four code smell types; Blob (God Class), Data Class, Long Method, and Feature Envy because they are among the most widely investigated code smells in the literature [62], [63], [69], [71], [72], [74], [79], [80]. MLCQ[1] dataset was used to detect these code smell for

[1] https://zenodo.org/records/3666840?preview_file=MLCQCodeSmellSamples.csv

several reasons. First, MLCQ was constructed through expert human validation by professional software developers using industry-relevant open-source projects [84], ensuring high-quality ground truth labels compared to datasets built through automated tool annotation. Second, MLCQ includes smell severity levels and exact source code locations, enabling fine-grained evaluation of detection accuracy. Third, MLCQ has been adopted in recent code smell detection studies such as [69], [71], allowing comparison of the thesis's findings. Finally, MLCQ's established reproducibility across the research community provides a strong basis for validating this experiment.

- Security Vulnerability Dataset: The Juliet Test Suite Java 1.3[2] was used to assess vulnerability detection capabilities for multiple reasons. First, it is a widely recognized benchmark dataset developed by the National Institute of Standards and Technology (NIST), providing well-defined and labeled test cases. Second, the dataset contains both vulnerable and non-vulnerable implementations, enabling balanced and controlled experimental evaluation. Third, Juliet covers a broad range of OWASP Top 10–related vulnerabilities making it suitable for security-focused empirical studies. Finally, Juliet has also been adopted in previous vulnerability detection studies [17], [68], [78], [82] allowing comparison of the thesis's findings.

6.2 Experimental Setup

Each selected test case was analyzed using ChatGPT (Codex 5.5 or GPT-5.5), Claude (Claude Sonnet 4.6), and a non-AI version of SonarQube as the traditional static analysis tool. ChatGPT and Claude were selected because previous studies have already evaluated earlier versions of these LLMs in both code smell detection

[2]https://samate.nist.gov/SARD/test-suites/111

and software vulnerability analysis [17] [68] [85]. Therefore, using their newer versions in this study enables an assessment of whether the capabilities of LLM-based code analysis tools have improved over time in terms of detection accuracy and issue coverage.

SonarQube was selected as the baseline automated static analysis tool because it is one of the most widely used tools in both industry and open-source software development [86]. In addition, it has been extensively used within projects of the Apache Software Foundation since 2019[3]. The tool provides automated detection of reliability, maintainability, and security issues, making it a suitable baseline for this study. Furthermore, it claims that reliability and maintainability rules are designed to produce zero false positives[4], as demonstrated by the results of this study in Table 6.3. The results produced by SonarQube do not directly correspond to code smell categories; instead, they are based on a set of predefined rule violations that approximate code quality and maintainability issues. Its rule-based analysis approach provides a strong benchmark for comparison against AI-driven methods. All tools were executed on identical code samples, and their outputs were systematically compared against the ground truth labels provided in the datasets to evaluate detection performance and consistency.

6.2.1 Evaluation Metrics and Confusion Matrix

Each tool's predictions were compared against the ground truth labels provided by the selected datasets. To evaluate tool effectiveness, this experiment employs four standard classification metrics; accuracy, precision, recall, and F1-score as they are widely adopted in prior work on AI-assisted code review, vulnerability detection, and code smell analysis [17], [60], [62], [63], [65], [67], [68], [69], [71], [74], [80], [81],

[3]https://sonarcloud.io/organizations/apache/projects

[4]https://docs.sonarsource.com/sonarqube-server/quality-standards-administration/managing-rules/rules

[82]. Precision and recall are particularly critical in software quality evaluation since false positives waste developer effort by flagging clean code, while false negatives undermine tool reliability by missing real issues. Both directly affect developer trust and the practical adoption of automated analysis tools. In addition, the F1-score balances these two concerns into a single measure, making it especially useful when comparing tools with different precision-recall trade-offs.

A confusion matrix was used to analyze and compare the classification performance across tools, providing a detailed breakdown of prediction outcomes by categorizing results into true positives, true negatives, false positives, and false negatives, as shown in Table 6.1.

Ground Truth	**Predicted Positive**	**Predicted Negative**
Positive (Smelly)	True Positive (TP)	False Negative (FN)
Negative (Non-smelly)	False Positive (FP)	True Negative (TN)

Table 6.1: Confusion matrix for evaluation

6.2.2 Code Smell Detection

The MLCQ dataset contains labeled instances representing four code smell types: Blob, Data Class, Feature Envy, and Long Method, collected from real-world open-source Java projects. Due to time constraints, a subset of the dataset was selected manually for empirical evaluation. However, the test set was constructed as a balanced dataset comprising 264 samples, including 132 positive (smelly) and 132 negative (non-smelly) instances. In addition, the samples are distributed across the four smell categories, as shown in Table 6.2. Blob and Data Class each contain 64 instances, while Feature Envy and Long Method each contain 68 instances. This distribution ensures that all smell types are adequately represented in the evaluation rather than emphasizing a particular category. Furthermore, each smell category includes instances with minor, major, and critical severity levels. Incorporating

multiple severity levels enables a more comprehensive assessment of the detection capabilities across varying degrees of code quality degradation.

Smell Type	**Positive**	**Negative**	**Total**
Blob	35	29	64
Data Class	35	29	64
Feature Envy	33	35	68
Long Method	29	39	68
Total	132	132	264

Table 6.2: Distribution of code smell samples

Codex 5.5 was selected for code smell detection instead of general purpose ChatGPT model because it is a code-specialized large language model optimized for software engineering tasks, making it more suitable for source code evaluation[5]. Table 6.3 presents the overall confusion matrix results for Codex 5.5, Claude Sonnet 4.6, and SonarQube. Among the evaluated tools, Claude Sonnet 4.6 achieved the highest number of true positives, indicating stronger detection capability; however, it also produced the highest number of false positives, suggesting a greater tendency to incorrectly classify non-smelly classes as smelly. Codex 5.5 achieved a comparable number of true positives while generating fewer false positives, demonstrating a more balanced trade-off between detection capability and classification accuracy. In contrast, SonarQube produced no false positives, indicating highly conservative behavior. However, this came at the expense of a large number of false negatives, meaning that many actual code smells were not detected.

Claude Sonnet 4.6 achieves the best overall performance in terms of F1-score (0.745) and shows particularly strong results for Feature Envy (F1 = 0.836) and Long Method (F1 = 0.842), as shown in 6.5. Codex 5.5 performs best on Data Class (F1 = 0.833), where it also achieves perfect precision, and it records the highest overall precision among the three tools (0.930), indicating a low rate of

[5]https://openai.com/index/introducing-codex/

Tool	TP	FP	TN	FN
Codex 5.5	80	6	126	52
Claude Sonnet 4.6	86	13	119	46
SonarQube	20	0	132	112

Table 6.3: Code Smell - Overall Confusion Matrix Results

Smell Type	Tool	TP	FP	TN	FN
Blob	Codex 5.5	14	0	29	21
	Claude Sonnet 4.6	17	0	29	18
	SonarQube	8	0	29	27
Data Class	Codex 5.5	25	0	29	10
	Claude Sonnet 4.6	17	3	26	18
	SonarQube	0	0	29	35
Feature Envy	Codex 5.5	21	5	30	12
	Claude Sonnet 4.6	28	6	29	5
	SonarQube	0	0	35	33
Long Method	Codex 5.5	20	1	38	9
	Claude Sonnet 4.6	24	4	35	5
	SonarQube	12	0	39	17

Table 6.4: Confusion Matrix Results by Code Smell Type

false positives. In contrast, SonarQube demonstrates perfect precision across all categories, meaning it does not produce false positives; however, this comes at the cost of extremely low recall (0.152). As a result, its overall F1-score is the lowest (0.263), and it completely fails to detect certain smell types such as Data Class and Feature Envy, where its F1-score drops to zero. Overall, the results suggest that the LLM-based approaches provide substantially broader code smell detection coverage than the traditional static analysis approach, although with an increased risk of false positive classifications.

Table 6.2.2 shows severity detection accuracy on true positive samples of LLMs. Although both LLM-based tools demonstrated reasonable detection accuracy, their

Tool	Precision	Recall	Accuracy	F1-score
Codex 5.5	**0.930**	0.606	**0.780**	0.734
Claude Sonnet 4.6	0.869	**0.652**	0.777	**0.745**
SonarQube	**1.000**	0.152	0.576	0.263

Table 6.5: Code Smell - Overall Performance Metrics

Smell Type	Tool	Precision	Recall	Accuracy	F1-score
Blob	Codex 5.5	1.000	0.400	0.672	0.571
	Claude Sonnet 4.6	1.000	**0.486**	**0.719**	**0.654**
	SonarQube	1.000	0.229	0.578	0.372
Data Class	Codex 5.5	**1.000**	**0.714**	**0.844**	**0.833**
	Claude Sonnet 4.6	0.850	0.486	0.672	0.618
	SonarQube	0.000	0.000	0.453	0.000
Feature Envy	Codex 5.5	0.808	0.636	0.750	0.712
	Claude Sonnet 4.6	**0.824**	**0.848**	**0.838**	**0.836**
	SonarQube	0.000	0.000	0.515	0.000
Long Method	Codex 5.5	0.952	0.690	0.853	0.800
	Claude Sonnet 4.6	**0.857**	**0.828**	**0.868**	**0.842**
	SonarQube	1.000	0.414	0.750	0.585

Table 6.6: Performance Metrics by Code Smell Type

severity classification performance was notably weaker. Both Claude Sonnet 4.6 and Codex 5.5 correctly detected the code smell in true positive cases but assigned the wrong severity level in more than half of those cases overall, with Claude Sonnet 4.6 achieving a severity accuracy of 48.8% and Codex 5.5 41.3%, indicating that detection capability does not imply reliable severity reasoning. Minor severity issues proved the most challenging for both tools; Codex 5.5 nearly completely failed at this level with only 4.3% accuracy, while Claude managed 30.8%, suggesting that both tools tend to overestimate the severity of minor issues. At the critical severity level, Claude Sonnet 4.6 performed considerably better than Codex 5.5, correctly classifying 63.0% of critical cases compared to 37.5%, indicating stronger reasoning about high-impact issues. Conversely, Codex 5.5 outperformed Claude Sonnet 4.6 at the Major severity level, achieving 69.7% accuracy compared to Claude's 51.5%. Overall, while Claude Sonnet 4.6 edges ahead of Codex 5.5 in severity classifica-

Severity	Codex TP	Codex Correct	Sonnet TP	Sonnet Correct
Critical	24	9 (37.5%)	27	17 (63.0%)
Major	33	23 (69.7%)	33	17 (51.5%)
Minor	23	1 (4.3%)	26	8 (30.8%)
Total	**80**	**33 (41.3%)**	**86**	**42 (48.8%)**

Table 6.7: Severity detection accuracy on true positive samples

tion, neither tool demonstrated sufficient reliability for severity assessment without human validation.

6.2.3 Vulnerability Detection

Due to the large size of the Juliet Test Suite, this experiment focuses on a subset of 19 CWE categories, listed in Table 6.2.3. Among these, three CWEs (CWE-79, CWE-89 and CWE-476) have a direct mapping to the MITRE Top 25 Most Dangerous Software Weaknesses[6]. Several vulnerabilities were selected from related categories within the MITRE Research Concepts hierarchy (View-1000). This approach was adopted because some weaknesses included in the MITRE Top 25 list, such as CWE-20 (Improper Input Validation) and CWE-22 (Path Traversal), are not directly represented in the Juliet Test Suite. Therefore, related weaknesses available in Juliet are selected as representative instances of these broader categories. For example, CWE-23 (Relative Path Traversal) was selected as a child weakness of CWE-22, and CWE-209 (Generation of Error Message Containing Sensitive Information) as a child of CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor). In addition, four CWEs (CWE-606, CWE-566, CWE-549, and CWE-523) were specifically selected based on the prior study [17] that reported detection challenges for ChatGPT and Claude in identifying these vulnerability types. Furthermore, four vulnerabilities were selected to provide coverage of credential management, information disclosure and cryptographic misuse issues, including CWE-256, CWE-259,

[6]https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

CWE-319 and CWE-327. The list of CWEs used in the experminent mentioned in Table 6.2.3.

No.	CWE ID	Name
1	CWE-23	Relative Path Traversal
2	CWE-36	Absolute Path Traversal
3	CWE-78	OS Command Injection
4	CWE-80	Basic Cross-site Scripting
5	CWE-89	SQL Injection
6	CWE-129	Improper Validation of Array Index
7	CWE-134	Use of Externally Controlled Format String
8	CWE-190	Integer Overflow
9	CWE-209	Generation of Error Message Containing Sensitive Information
10	CWE-226	Sensitive Information Uncleared Before Release
11	CWE-256	Plaintext Storage of a Password
12	CWE-259	Hard-coded Password
13	CWE-319	Cleartext Transmission of Sensitive Information
14	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
15	CWE-476	NULL Pointer Dereference
16	CWE-523	Unprotected Transport of Credentials
17	CWE-549	Missing Password Field Masking
18	CWE-566	Authorization Bypass Through User-Controlled SQL Primary Key
19	CWE-606	Unchecked Input for Loop Condition

Table 6.8: List of CWEs used in the experiment

For each selected CWE category, 5 vulnerable and 5 non-vulnerable samples were included, resulting in a total of 190(19*10) samples across all categories. Each category contains test cases of varying complexity levels to evaluate the effectiveness of detection tools such as multi-file test cases in which program logic is distributed across multiple related classes. These cases were selected to assess the ability of selected tools to analyze inter-file data flow and handle modular code structures. Before performing the analysis with LLMs, the original test case code was pre-processed to improve evaluation neutrality. This involved renaming package names, namespaces, class names, function names, and variables, as the original Juliet test case naming convention is highly descriptive and could potentially bias the model's interpretation. By replacing these identifiers with non-explanatory names and re-

moving comments, the study ensured that the LLM-based analysis focused solely on the actual code logic and behavior rather than semantic hints from the naming structure, thereby improving the validity of the experimental results.

Table 6.9 showed that none of the evaluated tools produced false positives for the selected test dataset. In the case of the LLM-based tools, some responses provided only partial or weaker mitigations; however, these cases were classified as true negatives because the target vulnerability was successfully mitigated. It has been observed that safe samples in the selected samples are explicit and contain clear remediation patterns, making them easier for modern LLMs to classify correctly. SonarQube occasionally reported alternative security issues in non-vulnerable samples; since the target CWE was not detected and the sample remained free of the evaluated vulnerability, these cases were also classified as true negatives (TN).

Claude Sonnet 4.6 achieved the highest CWE coverage by missing only 2 out of 19 selected CWE categories, followed by GPT 5.5, which missed 3 categories, while SonarQube showed the weakest coverage by failing to detect 15 CWE categories. The results further indicated that the LLMs were capable of identifying vulnerabilities across multiple complexity levels. In contrast, SonarQube demonstrated reduced effectiveness on multi-file cases and primarily detected vulnerabilities in simpler implementations. An exception was observed for CWE-256, which SonarQube successfully identified across all evaluated complexity levels.

It was also observed that GPT-5.5 predicted related but not exact CWE identifiers. For example, CWE-522 instead of CWE-549 and CWE-22 instead of CWE-23. This result was considered a true positive because CWE-522 and CWE-22 are parent of CWE-549 and CWE-23 respectively within the CWE research view(View-1000)[7].

In addition to the code-level analysis, the configuration file of one of my ASP.NET projects was also evaluated using both LLM. The model identified several security-

[7]https://cwe.mitre.org/data/definitions/522.html

Tool	**TP**	**FP**	**FN**	**TN**
GPT-5.5	79	0	16	95
Claude Sonnet 4.6	85	0	10	95
SonarQube	11	0	84	95

Table 6.9: Security Vulnerability - Confusion Matrix Results

Tool	**Precision**	**Recall**	**Accuracy**	**F1-score**
GPT-5.5	1.000	0.832	0.916	0.908
Claude Sonnet 4.6	1.000	**0.895**	**0.947**	**0.944**
SonarQube	1.000	0.116	0.558	0.208

Table 6.10: Security Vulnerability - Performance Metrics

related issues, including sensitive information exposure, missing Content Security Policy (CSP) directives, usage of outdated libraries, and absence of the HSTS (HTTP Strict Transport Security) header but it initially missed the issue of directory browsing. When directory browsing was explicitly queried using prompt (Is there any directory browsing vulnerability in this config file?), the model was able to correctly identify the issue.

Although directory browsing is typically enforced at the server level (e.g., IIS configuration) in ASP.NET applications, its consideration at the application level still contribute to defense-in-depth if server-side controls are misconfigured or absent. This observation highlights an important limitation, LLM outputs should not be treated as fully reliable security assessments [17]. Instead, they should be validated using developer expertise and complementary tools. In this case, the model initially missed an issue that is relevant in real-world security configurations, reinforcing the need for human verification in LLM-assisted security analysis.

Furthermore, studies show that using a specific prompt describing the smell performs 2.54 times [80]. It is also observed in the experiment that the effectiveness of the LLM-based analysis is highly dependent on the quality and specificity of the prompt. When a general prompt was used, the model highlighted only basic security concerns. After refining the prompt to explicitly include specific terms,

LLMs highlighted missed issues.

7 Discussion

7.1 Key Findings

LLM-based approaches consistently demonstrated stronger detection capability than the traditional static analysis tool across both code smell and vulnerability detection tasks. While SonarQube exhibited highly conservative behaviour, this came at the cost of substantially lower detection coverage than LLMs. LLMs also showed the ability to analyse code beyond simple single-file examples. Both models successfully identified issues in multi-file test cases where relevant program logic was distributed across multiple classes, suggesting an ability to reason about inter-file relationships and data flow. In contrast, SonarQube showed reduced effectiveness when analyzing multi-file test cases, failing to detect vulnerabilities in the majority of such cases.

Prompt quality and length were found to influence detection effectiveness. In particular, more specific prompts that explicitly described the target smell or vulnerability category produced more comprehensive and accurate results than general-purpose prompts, prompts are mentioned in Appendix B. Although the LLMs generally identified the correct vulnerability categories, they occasionally reported related parent CWE identifiers rather than the exact CWE assigned to the test case. This indicates that the models often understood the underlying security issue even when the precise CWE classification differed.

A notable finding of this study is that both Codex 5.5 and Claude Sonnet 4.6

demonstrated weak severity classification in code smell detection, with overall accuracies of 41.3% and 48.8%, respectively, and particularly poor performance on minor severity issues (4.3% for Codex and 30.8% for Claude). This observation reinforces the importance of combining LLM-generated findings with developer expertise and complementary security analysis techniques rather than relying solely on LLM outputs. Another important finding of this study is that no false positives were observed in the vulnerability detection experiments across all evaluated tools. This outcome may be explained by several factors acting in combination. The LLMs used in this study are more recent and demonstrate improved code understanding capabilities compared to those evaluated in earlier studies, and they also tend to adopt a more conservative approach by requiring stronger evidence before reporting a vulnerability. In addition, the structured nature of the Juliet Test Suite, which consists of well-defined synthetic test cases, may have contributed to easier and more reliable classification than the complex, multi-file real-world codebases. The relatively limited size of the evaluation dataset may also have reduced the likelihood of encountering false positive behaviour during testing. Furthermore, prompt design also influenced the results, as the use of explicit and structured prompts likely encouraged more cautious and evidence-based reasoning when identifying vulnerabilities.

During the comparative evaluation of LLMs, it was additionally observed that the code-specific model Codex 5.5 performed more reliably than the general-purpose GPT-5.5 model for code smell detection. In particular, several non-smelly Feature Envy instances that were incorrectly classified as false positives by GPT-5.5 were correctly identified as true negatives by Codex 5.5. This pattern was consistently observed across five samples during testing, suggesting that domain-specialized models may provide more precise discrimination in certain code smell categories compared to general-purpose LLMs.

7.2 Comparison with Literature Findings

7.2.1 Code Smells

Machine learning-based classifiers in previous studies achieved higher F1-scores compared to heuristic methods [73]. Overall, the findings of this study also indicate that LLM-based approaches outperform static analysis tools in terms of recall, accuracy, and F1-score. Claude Sonnet 4.6 achieved the highest overall F1-score (0.745) and recall (0.652), while Codex 5.5 achieved the precision (0.930). These results suggest that modern LLMs are capable of detecting code smells with performance comparable to previously reported AI-based approaches, although they do not yet reach the near-perfect results reported by task-specific deep learning models trained directly on code smell datasets, such as 100% accuracy for Long Method [74], all 4 performance metrics exceeding 90% for Long Method and Blob [69], and 98.34% accuracy for God Class in federated learning settings [62].

A particularly notable finding is that Claude Sonnet 4.6 achieved an F1-score of 0.836 for Feature Envy and 0.842 for Long Method, demonstrating strong performance in detecting method-level smells. Similarly, Codex 5.5 achieved its best performance on Data Class detection, with an F1-score of 0.833 and perfect precision (1.000), substantially outperforming the Data Class F-measure of 0.59 reported for GPT-3.5 Turbo by Silva et al. [80]. This suggests that recent generations of LLMs have improved their ability to reason about software design quality. Furthermore, SonarQube failed to detect any instances of Data Class and Feature Envy in the evaluation dataset. This observation aligns with several previous studies [63], [66], [73], which report that AI-based approaches are more effective at identifying complex code smells that require contextual understanding.

However, the results also confirm limitations previously reported in the literature. Consistent with the findings of Sadik et al. [66], the LLM-based approaches in this

study exhibit a trade-off between precision and recall. In our experiments, Claude Sonnet 4.6 achieves higher recall but produces more false positives, whereas Codex is more conservative and achieves higher precision. This behaviour suggests that, although LLMs provide broader detection coverage than static analysis tools, they remain prone to inconsistent classifications and require human verification before practical use.

While specialized machine learning and deep learning models achieve higher performance when trained specifically for individual smell types, modern LLMs offer the additional advantage of detecting multiple smell categories using a single general-purpose model without task-specific retraining.

7.2.2 Vulnerability Detection

The motivation for selecting the prior study [17] is that it evaluated GPT-4 and Claude 3 Opus using the same Juliet Test Suite dataset, enabling direct comparison with the current experiment. The previous study reported that these models of LLMs were unable to correctly identify CWEs 523, 549, 606, and 566. In contrast, the results of this experiment indicate that newer models, namely GPT-5.5 and Claude Sonnet 4.6, demonstrate improved detection capabilities by successfully identifying two of these previously challenging vulnerabilities while still struggling with others. GPT-5.5 correctly identified CWE-549 and CWE-606 but failed to detect CWE-523 and CWE-566. In one instance, GPT-5.5 recognized a related security issue but incorrectly classified CWE-523 as CWE-319, resulting in a false negative. Similarly, Claude Sonnet 4.6 successfully detected CWE-523 and CWE-549 but missed CWE-566 and CWE-606. For CWE-606, the model identified a security weakness but incorrectly mapped it to CWE-20 rather than the target CWE, and the result was therefore classified as a false negative. These findings suggest that newer LLMs have improved their ability to detect certain vulnerability categories;

however, challenges remain in accurately classifying closely related CWEs.

In addition, SonarQube failed to detect all four of these CWEs, consistent with findings from previous studies where other static analysis tools such as CodeQL and SpotBugs also failed to identify these vulnerabilities. Furthermore, LLM-based tools demonstrated strong performance in detecting issues such as hardcoded passwords, plaintext storage, and hardcoded cryptographic keys, consistent with prior research [65], [77]. SonarQube also detected these issues; however, its effectiveness was primarily limited to simpler code samples and reduced significantly with increasing code complexity.

Overall, the findings of the experiment are consistent with previous research [17] indicating that Claude Sonnet 4.6 performs better in providing clear and accurate vulnerability assessment verdicts. It followed the required response format more consistently and delivered definitive classifications. In contrast, GPT-5.5 occasionally produced ambiguous responses "POSSIBLY" rather than providing a direct determination. In some cases, additional prompting was required to obtain an explicit yes-or-no verdict. In our experiments, while both models demonstrate strong vulnerability detection capabilities, Claude Sonnet 4.6 provides more consistent and decisive assessments in code review tasks.

The experiment result aligns with the broader finding in the literature that SonarQube's reliance on manually configured rule sets and pattern matching limits its ability to detect more abstract design flaws, whereas LLMs offer a more flexible, semantic-based approach capable of identifying a broader range of architectural and design issues [66]. Consequently, software teams should position LLMs as the primary tool for broad smell discovery, particularly for design-level and behavioural smells, while reserving static analysis tools like SonarQube for deterministic, syntax-level checks where their zero false positive rate is most valuable [66]. In addition, human reviewers should remain responsible for validating findings, resolving ambigu-

ous cases, and making final decisions regarding code quality and security. Such a hybrid workflow combines the contextual reasoning capabilities of AI-assisted tools with the precision and consistency of traditional static analysis, leading to more effective and reliable software quality assurance.

7.3 Limitations and Threat to Validity

This study has several limitations that should be considered when interpreting the findings. The study focuses on the four most widely studied and commonly reported code smells in the literature, but the results may not generalize to other types of code smells. Similarly, the vulnerability detection experiment covered a broad range of important security weaknesses, including vulnerabilities related to the MITRE Top 25 list and their associated CWE categories, but it did not encompass all possible software vulnerabilities. Consequently, the findings should not be interpreted as a comprehensive assessment of AI-assisted tools across the entire vulnerability landscape.

The evaluation was conducted using benchmark datasets rather than real-world industrial codebases. The benchmark datasets may not fully capture the complexity, scale, and contextual dependencies found in production software systems; therefore, tool performance in real-world environments may differ from the results reported in this study. Moreover, the subset of samples is relatively small compared to the full MLCQ and Juliet datasets. Although the selected samples were balanced across code smell categories, severity levels, and vulnerable/non-vulnerable instances, the results should be interpreted as exploratory rather than statistically generalizable. The evaluation was limited to Java source code. While Java is widely used in both academic research and industrial practice, the findings may not generalize to other programming languages with different syntax, paradigms, and security characteristics. Although no false positives were observed during vulnerability detection, this

result should be interpreted with caution. The absence of false positives may be influenced by the selected samples, dataset characteristics, prompt design, and evaluation criteria. Previous studies have reported false positives, hallucinations, and vulnerability misclassification in LLM-based code analysis [17], [81], [82]. Therefore, the findings of this study do not imply that modern LLMs are free from false positives; rather, they indicate that no false positives were observed within the specific experimental setting used in this evaluation.

In addition, the results are subject to prompt dependency, a commonly reported limitation in AI-assisted code review research. Different prompt formulations may influence the responses generated by LLMs, potentially affecting detection performance and consistency. To mitigate this threat, the same prompt template was used across all evaluated samples. AI models evolve rapidly, and future versions of Codex, ChatGPT, Claude, or other LLMs may exhibit different capabilities and performance characteristics. Consequently, the reported results should be interpreted as a snapshot of model performance at the time of evaluation rather than a definitive assessment of AI-assisted code review tools. Furthermore, LLM configuration and fine-tuning were outside the scope of the experiments. They were conducted using publicly available versions of Codex, ChatGPT, and Claude. As these services manage parameters such as temperature, reasoning strategies, and other inference settings internally, these configurations could not be explicitly controlled or reproduced. This limitation may affect experimental reproducibility and introduce variability that cannot be fully quantified by the researcher.

8 Conclusion

8.1 Summary

This thesis investigated the effectiveness of AI-assisted code review tools for detecting code smells and security vulnerabilities and compared their performance with traditional static analysis approaches. A systematic literature review was conducted to examine recent research on AI-assisted code review, followed by an exploratory empirical evaluation using large language models (Codex 5.5, GPT-5.5 and Claude Sonnet 4.6) and the static analysis tool SonarQube. The literature review showed that AI-assisted approaches, including machine learning, deep learning, and large language models, generally achieve strong performance in software quality analysis. Previous studies reported improvements in accuracy, precision, recall, and F1-score for both code smell detection and vulnerability identification.

The empirical evaluation largely supported the findings of the literature review. For code smell detection, the experiment indicates that LLM-based approaches are more effective at identifying a broader range of maintainability issues, whereas static analysis tools remain conservative and may miss many smell instances. For security vulnerability detection, LLMs successfully identified most vulnerabilities across the selected CWE categories. SonarQube maintained high precision but detected considerably fewer vulnerabilities. These findings suggest that LLMs can complement traditional static analysis by providing broader semantic understanding and

improved detection coverage. Although no false positives were observed in the vulnerability detection, this finding should be interpreted cautiously and does not imply that modern LLMs are inherently free from false-positive detections.

Overall, the results indicate that AI-assisted code review tools can substantially enhance software quality assurance by improving the detection of both maintainability and security issues. However, they should not be viewed as replacements for traditional static analysis or human expertise. The experimental results revealed that despite strong detection performance, LLMs exhibited notable weaknesses in severity classification, correctly assigning severity in fewer than half of the true positive cases. Furthermore, LLMs are inherently non-deterministic, repeated queries on identical code may produce inconsistent outputs, which undermines reproducibility in automated testing pipelines. The tendency of LLMs to generate hallucinated or partially correct responses, as observed in this study and reported by [82] and [67], further limits their reliability as standalone tools. Additionally, LLMs failed entirely on certain vulnerability categories such as Authorization Bypass (CWE-566), and demonstrated limited reasoning, indicating that semantic understanding remains bounded. On the other hand, traditional static analysis tools such as SonarQube, while limited in detection breadth, offer deterministic, reproducible, and auditable results that are essential in compliance-driven development environments. Instead, the literature and the experimental findings support a complementary approach in which AI-assisted tools are used for broad initial discovery, static analysis provides deterministic baseline checks, and human review is applied for severity validation, complex judgements, and final remediation decisions, together achieving more reliable and comprehensive code quality assessment.

8.2 Answers to Research Questions

RQ1: How do AI-assisted code review tools perform compared to traditional static analysis in terms of defect detection accuracy and efficiency?

In the empirical evaluation of this thesis, LLM-based tools consistently outperform static analysis in terms of recall, accuracy, and F1-score. In code smell detection, Claude Sonnet 4.6 achieved the highest overall F1-score (0.745) and recall (0.652), while Codex 5.5 achieved the highest accuracy (0.780). SonarQube acheived the highest precision (1.000) but with lowest recall (0.152). Similarly in vulnerability detection, Claude Sonnet 4.6 achieved the highest recall at 0.895 with an F1 of 0.944, followed by GPT-5.5 at 0.832 with an F1 of 0.908, while SonarQube achieved only 0.116 recall with an F1 of 0.208 across 19 CWEs and 190 test samples. Both LLMs maintained a zero false positive rate, whereas SonarQube's detection was limited to pattern-recognisable vulnerabilities such as hard-coded passwords and weak cryptography.

RQ2: What types of software quality issues, such as code smells and security vulnerabilities, are most commonly detected by AI-assisted code review tools compared to traditional approaches?

In this study, LLMs effectively detect both code smells and security vulnerabilities, demonstrating highest F1-measure in method-level smells (Feature Envy and Long Method) and consistent detection across 17 out of 19 CWE categories. In contrast, static analysis tools fail to detect certain code smell types, including Data Class and Feature Envy, and miss 16 CWE categories overall, while also showing reduced coverage in multi-file vulnerability scenarios. Common issues detected by AI-based tools reported in previous studies are discussed in detail in Section 5.3.

RQ3: What challenges and limitations are reported in existing studies regarding the reliability, interpretability, and practical adoption of AI-assisted code review tools?

The literature consistently highlights several key limitations of AI-assisted code review tools. A dominant issue across studies is the precision–recall trade-off, where improving recall often leads to more false positives, and increasing precision reduces detection coverage [66]. This behaviour is also observed in this study when detecting code smell, where Claude Sonnet 4.6 achieves higher recall but generates more false positives, while Codex 5.5 is more conservative with higher precision.

Another widely reported challenge is prompt dependency and model sensitivity, particularly in LLM-based approaches [80], [81]. It is also observed in the experiment that performance varies significantly depending on prompt design, input formatting, and context size. In addition, several studies report context limitations and hallucination risks, especially when analyzing large or multi-file codebases [82]. Therefore, each code sample was evaluated individually during the experiment to ensure that the entire source code could be processed within the model's available context.

Scalability and computational cost are also recurring concerns, particularly for deep learning and hybrid systems requiring extensive training or large model inference. Other limitations include interpretability issues, data imbalance sensitivity, and difficulty in localization of defects, as reported in multiple studies [60], [76], [78]. The empirical findings of this thesis align with these limitations. While LLMs provide strong detection performance and broader coverage than static analysis tools, they still produce inconsistent classifications and require human verification before practical adoption. SonarQube's extremely low recall further highlights the practical trade-off between precision-oriented static rules and recall-oriented AI-based methods.

Overall, the findings suggest that while AI-assisted code review tools and LLM-based approaches have achieved substantial improvements in detection capability compared to traditional static analysis tools, their practical adoption in industrial settings is still constrained by issues of reliability, inconsistent outputs, and lim-

ited trust. In particular, the trade-offs between precision and recall, sensitivity to prompts, and hallucination risks indicate that current systems are not yet sufficiently stable for fully automated use. As a result, these tools are best positioned as decision-support systems rather than standalone solutions, requiring human oversight to ensure dependable code review outcomes.

8.3 Future Work

Future research should extend the evaluation to larger and more diverse datasets, including real-world open-source projects, to assess whether the detection performance of LLMs generalizes beyond synthetic benchmarks. Another important direction is the investigation of fine-tuning approaches. Future studies should explore whether fine-tuning LLMs can improve detection accuracy, reduce severity misclassification, and lower hallucination rates. Prior work by [83] shows that fine-tuned GPT-3.5 achieved a 73–74% improvement in exact match scores in code review tasks, suggesting that domain-specific fine-tuning may also be beneficial for vulnerability and code smell detection.

Future research should also place emphasis on LLM-based severity assessment as a critical component of code smell and vulnerability analysis. While most existing studies primarily focus on detecting the presence of issues, severity assessment is equally important for understanding their practical impact in real-world software systems. Moreover, the current study focused exclusively on Java-based code samples. Therefore, future work should extend the evaluation to other widely used programming languages such as Python, JavaScript, C, and C++. This is important because vulnerability patterns and code smell manifestations often differ across languages, and LLM performance may vary depending on language-specific syntax and training data availability.

Another limitation identified in this study is the non-deterministic nature of

LLM outputs under default temperature settings, which can affect reproducibility. The impact of different temperature settings on result consistency should be examined. In addition, future research should evaluate the integration of hybrid pipelines combining LLMs and static analysis tools within real software development workflows. Several existing studies have reported performance improvements when AI-based tools are used in combination with traditional static analysis techniques [63], [65], [67]. Therefore, it is also important to assess whether newer LLMs improve performance when integrated into such hybrid approaches, and to what extent they enhance detection accuracy and reduce false classifications. Furthermore, developer-centered studies are needed to assess usability, trust, and adoption of AI-assisted code review systems. Recent advances in LLM-based software engineering, such as agentic frameworks and Retrieval-Augmented Generation (RAG), enable models to iteratively reason over large codebases, retrieve relevant context, and perform multi-step analysis. Future work should investigate whether these approaches mitigate the context limitations observed in this study and whether they provide measurable improvements over single-prompt evaluation methods. Additionally, the capabilities of newer and more advanced models should be systematically evaluated as they continue to evolve.

References

[1] J. Naulty, E. Chen, J. Wang, G. Digkas, and K. Chalkias, *Bugdar: Ai-augmented secure code review for github pull requests*, 2025. arXiv: 2503.17302 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2503.17302.

[2] B. Sodhi and S. Sharma, *Using stackoverflow content to assist in code review*, 2018. arXiv: 1803.05689 [cs.SE]. [Online]. Available: https://arxiv.org/abs/1803.05689.

[3] H. Lal and G. Pahwa, "Code review analysis of software system using machine learning techniques", in *2017 11th International Conference on Intelligent Systems and Control (ISCO)*, 2017, pp. 8–13. DOI: 10.1109/ISCO.2017.7855962.

[4] P. Garg and S. H. Sengamedu, "Example-based synthesis of static analysis rules", *ArXiv*, vol. abs/2204.08643, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248239618.

[5] A. Y. Gerasimov, "Survey on static program analysis results refinement approaches", 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:198453413.

[6] W. Olabiyi, D. Akinyele, and E. Joel, "The evolution of ai: From rule-based systems to data-driven intelligence", Jan. 2025.

[7] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Codegru: Context-aware deep learning with gated recurrent unit for source code modeling", *Information and*

Software Technology, vol. 125, p. 106 309, 2020, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106309. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300616.

[8] D. Mahalakshmi, P. Kasinathan, D. Elangovan, C. R. Bhat, M. Balamurugan, and S. Sivakumar, "Code smell detection using hybrid machine learning algorithms", in *2023 5th International Conference on Inventive Research in Computing Applications (ICIRCA)*, 2023, pp. 633–638. DOI: 10.1109/ICIRCA57980.2023.10220911.

[9] I. Medeiros, N. F. Neves, and M. P. Correia, "Dekant: A static analysis tool that learns to detect web application vulnerabilities", *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:17527825.

[10] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?", in *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, Italy: IEEE, Mar. 2018, pp. 612–621. DOI: 10.1109/SANER.2018.8330266.

[11] M. Fowler, "Refactoring - improving the design of existing code", in *Addison Wesley object technology series*, 1999. [Online]. Available: https://api.semanticscholar.org/CorpusID:263888396.

[12] P. B. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice", *IEEE Software*, vol. 29, pp. 18–21, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:10673584.

[13] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?", in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 306–315. DOI: 10.1109/ICSM.2012.6405287.

[14] M. Škipina, J. Slivka, N. Luburić, and A. Kovačević, "Automatic detection of feature envy and data class code smells using machine learning", *Expert Systems with Applications*, vol. 243, p. 122 855, 2024, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2023.122855. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423033572.

[15] T. Sharma and D. Spinellis, "A survey on software smells", *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.12.034. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217303114.

[16] A. Kovačević et al., "Automatic detection of long method and god class code smells through neural source code embeddings", *Expert Systems with Applications*, vol. 204, p. 117 607, 2022, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2022.117607. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422009186.

[17] K. Tamberg and H. Bahsi, "Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study", *IEEE Access*, vol. 13, pp. 29 698–29 717, 2025. DOI: 10.1109/ACCESS.2025.3541146.

[18] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning", *Electronic Markets*, vol. 31, no. 3, pp. 685–695, Apr. 2021, ISSN: 1422-8890. DOI: 10.1007/s12525-021-00475-2. [Online]. Available: http://dx.doi.org/10.1007/s12525-021-00475-2.

[19] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning: The mit press, 2016, 800 pp, isbn: 0262035618", *Genetic Programming and Evolvable Machines*, vol. 19, Oct. 2017. DOI: 10.1007/s10710-017-9314-z.

[20] N. Nilsson, *The quest for artificial intelligence: A history of ideas and achievements*. Jan. 2010, ISBN: 9780521122931. DOI: 10.1017/CBO9780511819346.

[21] T. Mitchell, *Machine Learning* (McGraw-Hill International Editions). McGraw-Hill, 1997, ISBN: 9780071154673. [Online]. Available: https://books.google.fi/books?id=EoYBngEACAAJ.

[22] C. Bishop, *Pattern Recognition and Machine Learning*, ISBN: 978-0-387-31073-2.

[23] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, *Deep learning based vulnerability detection: Are we there yet?*, 2020. arXiv: 2009.07235 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2009.07235.

[24] OpenAI et al., *Gpt-4 technical report*, 2024. arXiv: 2303.08774 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2303.08774.

[25] Z. Feng et al., "Codebert: A pre-trained model for programming and natural languages", Jan. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

[26] A. Vaswani et al., *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[27] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks", *Information Processing Management*, vol. 45, no. 4, pp. 427–437, 2009, ISSN: 0306-4573. DOI: https://doi.org/10.1016/j.ipm.2009.03.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306457309000259.

[28] R. Tufano and G. Bavota, *Automating code review: A systematic literature review*, 2025. arXiv: 2503.09510 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2503.09510.

[29] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review", in *Proceedings of the 2013 International Conference on Soft-*

ware Engineering, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 712–721, ISBN: 9781467330763.

[30] J. Cunha, G. Couto, A. Ribeiro, and J. Saraiva, "On the effectiveness of modern code review: Results from an empirical study", *Journal of Systems and Software*, vol. 176, p. 110 939, 2021.

[31] K. Nehéz and N. Khleel, "Tools, processes and factors influencing of code review", *Multidiszciplináris Tudományok*, vol. 10, pp. 277–284, Oct. 2020. DOI: 10.35925/j.multi.2020.3.33.

[32] H. Kim, Y. Kwon, S. Joh, H. Kwon, Y. Ryou, and T. Kim, "Understanding automated code review process and developer experience in industry", *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253421633.

[33] B. Gunawan and A. T. Sitorus, "Enhancing software quality through automated code review tools: An empirical synthesis across ci/cd pipelines", *Digitus : Journal of Computer Science Applications*, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:281987072.

[34] G. Kathiresan, "Exploring the role of large language models in automated code review and software quality enhancement", *International Journal of Innovative Research in Science, Engineering and Technology*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:277291681.

[35] R. Tufano, A. Martin-Lopez, A. Tayeb, O. Dabić, S. Haiduc, and G. Bavota, *Deep learning-based code reviews: A paradigm shift or a double-edged sword?*, 2024. arXiv: 2411.11401 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2411.11401.

[36] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, “A large empirical assessment of the role of data balancing in machine-learning-based code smell detection”, *Journal of Systems and Software*, vol. 169, p. 110 693, 2020, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110693. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301448.

[37] T. Sharma et al., *A survey on machine learning techniques for source code analysis*, 2022. arXiv: 2110.09610 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2110.09610.

[38] S. Zaharia, T. Rebedea, and S. Trausan-Matu, “Machine learning-based security pattern recognition techniques for code developers”, *Applied Sciences*, vol. 12, no. 23, p. 12 463, 2022. DOI: 10.3390/app122312463.

[39] U. Cihan, A. Içöz, V. Haratian, and E. Tüzün, “Evaluating large language models for code review”, *ArXiv*, vol. abs/2505.20206, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:278911858.

[40] Z. Rasheed et al., “Ai-powered code review with llms: Early results”, *ArXiv*, vol. abs/2404.18496, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:269449874.

[41] S. Goldman et al., “What types of code review comments do developers most frequently resolve?”, *ArXiv*, vol. abs/2510.05450, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:281886301.

[42] A. Alami and N. A. Ernst, “Human and machine: How software engineers perceive and engage with ai-assisted code reviews compared to their peers”, *2025 IEEE/ACM 18th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 63–74, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:275336938.

[43] Y. Almeida et al., "Aicodereview: Advancing code quality with ai-enhanced reviews", *SoftwareX*, vol. 26, p. 101 677, 2024, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2024.101677. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711024000487.

[44] B. Yetistiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt", *ArXiv*, vol. abs/2304.10778, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258291698.

[45] M. Ayyarrappan, "Ai for automated code reviews and quality assurance", *International Scientific Journal of Engineering and Management*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:277601283.

[46] M. S. S. Chowdhury, M. N. U. R. Chowdhury, F. F. Neha, and A. Haque, "Ai-powered code reviews: Leveraging large language models", in *2024 International Conference on Signal Processing and Advance Research in Computing (SPARC)*, vol. 1, 2024, pp. 1–6. DOI: 10.1109/SPARC61891.2024.10829223.

[47] N. A. Ernst, G. Bavota, and T. Menzies, "Ai-driven development is here: Should you worry?", *IEEE Software*, vol. 39, pp. 106–110, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:246831040.

[48] M. Vijayvergiya et al., "Ai-assisted assessment of coding practices in modern code review", in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, ser. AIware 2024, Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 85–93, ISBN: 9798400706851. DOI: 10.1145/3664646.3665664. [Online]. Available: https://doi.org/10.1145/3664646.3665664.

[49] D. Fried et al., *Incoder: A generative model for code infilling and synthesis*, 2023. arXiv: 2204.05999 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2204.05999.

[50] F. Caram, B. R. de Oliveira Rodrigues, A. Campanelli, and F. Silva Parreiras, "Machine learning techniques for code smells detection: A systematic mapping study", *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, pp. 285–316, Feb. 2019. DOI: 10.1142/S021819401950013X.

[51] K.-G. Grujic, S. Prokić, A. Kovačević, N. Luburić, D. Vidakovic, and J. Slivka, "Machine learning approaches for code smell detection: A systematic literature review", *SSRN Electronic Journal*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:254669891.

[52] T. Lewowski and L. Madeyski, "Code smells detection using artificial intelligence techniques: A business-driven systematic review", *Developments in Information & Knowledge Management for Business Applications*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:238693836.

[53] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms", *ACM Comput. Surv.*, vol. 55, no. 13s, Jul. 2023, ISSN: 0360-0300. DOI: 10.1145/3596908. [Online]. Available: https://doi.org/10.1145/3596908.

[54] K. Sun et al., "Does ai code review lead to code changes? a case study of github actions", *ArXiv*, vol. abs/2508.18771, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:280870178.

[55] Y. Wang, "A review of research on ai-assisted code generation and ai-driven code review", *Academic Journal of Science and Technology*, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:283508847.

[56] S. Ahmed, “Integrating ai-driven automated code review in agile development: Benefits, challenges, and best practices”, *International Journal of Advanced Engineering, Management and Science*, 2025. [Online]. Available: `https://api.semanticscholar.org/CorpusID:276919299`.

[57] F. S. Aðalsteinsson, B. B. Magnússon, M. Milicevic, A. N. Davidsson, and C.-H. Cheng, “Rethinking code review workflows with llm assistance: An empirical study”, *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 488–497, 2025. [Online]. Available: `https://api.semanticscholar.org/CorpusID:278788489`.

[58] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering”, vol. 2, Jan. 2007.

[59] S. Han, H. Nam, J. Kang, K. Kim, S. Cho, and S. Lee, “Code-smash: Source-code vulnerability detection using siamese and multi-level neural architecture”, *IEEE Access*, vol. 12, pp. 102 492–102 504, 2024. DOI: `10.1109/ACCESS.2024.3432323`.

[60] Y. Chen, “Autoreview: An llm-based multi-agent system for security issue-oriented code review”, 2025, pp. 1022–1024. DOI: `10.1145/3696630.3728618`. [Online]. Available: `https://www.scopus.com/inward/record.uri?eid=2-s2.0-105013966935&doi=10.1145%2f3696630.3728618&partnerID=40&md5=3d1aa7f9ee07192418f7a7fbce3030a6`.

[61] B. Steenhoek, H. Gao, and W. Le, “Dataflow analysis-inspired deep learning for efficient vulnerability detection”, in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24, Lisbon, Portugal: Association for Computing Machinery, 2024, ISBN: 9798400702174. DOI: `10.1145/3597503.3623345`. [Online]. Available: `https://doi.org/10.1145/3597503.3623345`.

[62] S. Alawadi et al., "Fedcsd: A federated learning based approach for code-smell detection", *IEEE Access*, vol. 12, pp. 44 888–44 904, 2024. DOI: 10.1109/ACCESS.2024.3380167.

[63] D. Zhang, S. Song, Y. Zhang, H. Liu, and G. Shen, "Code smell detection research based on pre-training and stacking models", *IEEE Latin America Transactions*, vol. 22, no. 1, pp. 22–30, 2024. DOI: 10.1109/TLA.2024.10375735.

[64] G. Ibba, R. Neykova, M. Ortu, R. Tonelli, S. Counsell, and G. Destefanis, "A machine learning approach to vulnerability detection combining software metrics and topic modelling: Evidence from smart contracts", *Machine Learning with Applications*, vol. 22, p. 100 759, 2025, ISSN: 2666-8270. DOI: https://doi.org/10.1016/j.mlwa.2025.100759. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666827025001422.

[65] J. A. Santas Ciavatta, J. R. Bermejo Higuera, J. Bermejo Higuera, J. A. S. Montalvo, T. S. Riera, and J. Pérez Melero, "Integration of large language models (llms) and static analysis for improving the efficacy of security vulnerability detection in source code", *Computers, Materials and Continua*, vol. 86, no. 3, 2026, ISSN: 1546-2218. DOI: https://doi.org/10.32604/cmc.2025.074566. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1546221826000603.

[66] A. R. Sadik and S. Govind, "Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3", 2025, pp. 969–975. DOI: 10.1145/3756681.3756993. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105027168509&doi=10.1145%2f3756681.3756993&partnerID=40&md5=da089c1483a72725479ccc48245e741d.

[67] P. J. Chapman, C. Rubio-González, and A. V. Thakur, "Interleaving static analysis and llm prompting with applications to error specification inference",

International Journal on Software Tools for Technology Transfer, vol. 27, no. 2, pp. 239–254, 2025. DOI: 10.1007/s10009-025-00780-7. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85218840766&doi=10.1007%2fs10009-025-00780-7&partnerID=40&md5=35ca83b0dc450ae8ece473a73b8e0860.

[68] Z. Feng, Y. Chen, K. Zhang, X. Li, and G. Liu, "Concurrency bug detection via static analysis and large language models", *Future Internet*, vol. 17, no. 12, 2025. DOI: 10.3390/fi17120578. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105025888648&doi=10.3390%2ffi17120578&partnerID=40&md5=33110ed7908ec4dd7231806765d08c5a.

[69] M. Zhang, J. Jia, L. F. Capretz, X. Hou, and H. Tan, "Graph neural network-based long method and blob code smell detection", *Science of Computer Programming*, vol. 243, p. 103 284, 2025, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2025.103284. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642325000231.

[70] W. Xu and X. Zhang, "Multi-granularity code smell detection using deep learning method based on abstract syntax tree", vol. 2021-July, 2021, pp. 503–509. DOI: 10.18293/SEKE2021-014. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85114279916&doi=10.18293%2fSEKE2021-014&partnerID=40&md5=4f914c5ed0bbf6d673686baa9098b413.

[71] "Ensesmells : Deep ensemble and programming language models for automated code smells detection", *Journal of Systems and Software*, vol. 224, p. 112 375, 2025, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2025.112375. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121225000433.

[72] A. Nızam, E. İslamoğlu, Ö. Kerem Adali, and M. Aydin, "Optimizing pre-trained code embeddings with triplet loss for code smell detection", *IEEE Access*, vol. 13, pp. 31 335–31 350, 2025. DOI: 10.1109/ACCESS.2025.3542566.

[73] E. Hamouda, A. El-Korany, and S. Makady, "Smell-ml: A machine learning framework for detecting rarely studied code smells", *IEEE Access*, vol. 13, pp. 12 966–12 980, 2025. DOI: 10.1109/ACCESS.2025.3530927.

[74] N. A. A. Khleel and K. Nehéz, "Improving accuracy of code smells detection using machine learning with data balancing techniques", *Journal of Supercomputing*, vol. 80, no. 14, pp. 21 048–21 093, 2024. DOI: 10.1007/s11227-024-06265-9. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85195272834&doi=10.1007%2fs11227-024-06265-9&partnerID=40&md5=b02b847fe3122b92e16dbc8b0914618a.

[75] C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang, "Knighter: Transforming static analysis with llm-synthesized checkers", 2025, pp. 655–669. DOI: 10.1145/3731569.3764827. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105020853239&doi=10.1145%2f3731569.3764827&partnerID=40&md5=a4c0a13f92d553551b13bef941512e89.

[76] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Vulnerable code detection using software metrics and machine learning", *IEEE Access*, vol. 8, pp. 219 174–219 198, 2020. DOI: 10.1109/ACCESS.2020.3041181.

[77] G. D. Vito, F. Palomba, and F. Ferrucci, "Secllm: Enhancing security smell detection in iac with large language models", *IEEE Access*, vol. 13, pp. 204 480–204 498, 2025. DOI: 10.1109/ACCESS.2025.3637505.

[78] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, "Vulnerability prediction from source code using machine learning", *IEEE Access*, vol. 8, pp. 150 672–150 684, 2020. DOI: 10.1109/ACCESS.2020.3016774.

[79] R. Gupta, N. Kumar, S. Kumar, and J. K. Seth, "Unsupervised machine learning for effective code smell detection: A novel method", *Journal of Communications Software and Systems*, vol. 20, no. 4, pp. 307–316, 2024. DOI: 10.24138/jcomss-2024-0083. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85212790196&doi=10.24138%2fjcomss-2024-0083&partnerID=40&md5=575188b81ff0ee6c7758fe1b2255c042.

[80] L. Silva, J. Silva, J. Montandon, and M. Valente, "Detecting code smells using chatgpt: Initial insights", Jul. 2024. DOI: 10.13140/RG.2.2.36634.04802.

[81] E. Woźny, J. Hryszko, and A. Roman, "Leveraging large language models for software defect detection", in *Software Engineering and Advanced Applications*, D. Taibi and D. Smite, Eds., Cham: Springer Nature Switzerland, 2026, pp. 125–142, ISBN: 978-3-032-04200-2.

[82] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, *Understanding the effectiveness of large language models in detecting security vulnerabilities*, 2024. arXiv: 2311.16169 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2311.16169.

[83] C. Pornprasit and C. Tantithamthavorn, "Fine-tuning and prompt engineering for large language models-based code review automation", *Information and Software Technology*, vol. 175, p. 107 523, 2024, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2024.107523. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924001289.

[84] L. Madeyski and T. Lewowski, "Mlcq: Industry-relevant code smell data set", in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '20, New York, NY, USA: Association for Computing Machinery, 2020, pp. 342–347, ISBN: 9781450377317. DOI:

10.1145/3383219.3383264. [Online]. Available: https://doi.org/10.1145/3383219.3383264.

[85] J. Bae, S. Kwon, and S. Myeong, "Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques", *Electronics*, vol. 13, no. 13, 2024, ISSN: 2079-9292. DOI: 10.3390/electronics13132657. [Online]. Available: https://www.mdpi.com/2079-9292/13/13/2657.

[86] V. Lenarduzzi, A. Sillitti, and D. Taibi, "A survey on code analysis tools for software maintenance prediction", in *Proceedings of 6th International Conference in Software Engineering for Defence Applications*, P. Ciancarini, M. Mazzara, A. Messina, A. Sillitti, and G. Succi, Eds., Cham: Springer International Publishing, 2020, pp. 165–175, ISBN: 978-3-030-14687-0.

Appendix A

Table A.1: Data Extraction of Previous Studies

ID	AI Technique	RQ2: Target Issue	RQ1 and RQ3
[74]	Bidirectional Long Short-Term Memory (Bi-LSTM) and Gated Recurrent Unit (GRU)	Code smells (God Class, Data Class, Feature Envy, Long Method)	RQ1: Improved accuracy with data balancing (100% for long method). RQ3: Generalizability, tuning challenges.
[69]	Graph Neural Network (GNN)	Code smells (Long Method, Blob)	RQ1: Long Method has achieved a remarkable 0.97, and the values of other performance metrics all exceed 0.9. RQ3: Class imbalance, complexity.
[71]	Deep Learning + embeddings	Code smells (Long Method, Feature Envy, Data Class, God Class)	RQ1: Improved accuracy from 5.98% to 28.26% via structural + semantic features. RQ3: Class imbalance, dataset dependency.

Continued on next page

ID	AI Technique	RQ2: Target Issue	RQ1 and RQ3
[80]	ChatGPT version 3.5-Turbo	Code smells (Blob, Data Class, Feature Envy, Long Method)	RQ1: With detailed prompt, main improvement occurred for Data Class where F-measure rose by 0.48 to reach 0.59. However, for all other smells, the F-measure remained below 0.50 RQ3: Prompt dependency and token window constraints.
[62]	Federated learning	Code smells (God Class)	RQ1: High accuracy (98.34%) and scalable. RQ3: Privacy, model drift.
[63]	BERT + static analysis	Code smells (Brain Class, Data Class, God Class, Brain Method)	RQ1: Improves the average accuracy by 10.38% compared to existing detection methods, while maintaining high precision, recall, and F1 scores. RQ3: Data imbalance, cost.
[72]	Contrastive learning	4 Code smells (Feature envy, long method, long parameter list, data class)	RQ1: Improves representation accuracy (+13%). RQ3: Cost, reproducibility issues.
[79]	Unsupervised ML	Code smells (Long Method, Feature Envy, God Class, Data Class)	RQ1: Close to supervised but lower accuracy. RQ3: Bias, generalization issues.

Continued on next page

ID	AI Technique	RQ2: Target Issue	RQ1 and RQ3
[73]	ML classifiers	5 Rarely code smells (Middle Man, Class Data, Inappropriate Intimacy, Refused Bequest, Speculative Generality)	RQ1: Higher F1 than heuristic methods. RQ3: Data imbalance, subjective labeling.
[66]	GPT-4.0 vs DeepSeek-V3	Code smells (bloaters, couplers, dispensables, object-orientation abusers and change preventers)	RQ1: GPT-4.0 higher precision (0.79) than DeepSeek-V3 (0.42); both showed low recall; GPT (0.41) and DeepSeek (0.31). RQ3: Precision–recall trade-off.
[70]	Deep learning approach	4 Code smells (Deficient Encapsulation, Insufficient Modularization, Feature Envy, Empty Catch Block)	RQ1: Improves F-measure (0.95).
[83]	GPT-3.5, Magicoder	Code review automation	RQ1: Fine-tuned GPT-3.5 improves EM(Exact Match) by 73–74%. RQ3: Training cost is high.
[65]	Multiple LLMs + Static Application Security Testing (SAST) tools	11 security vulnerabilities (XSS, SQL injection, Command Injection, Path Traversal, Insecure Cookie, Weak Encyption etc)	RQ1: In most of the cases, significant performance over standalone approaches with (Acc:0.90, Pre:0.92, Rec:0.88, F1:0.90). RQ3: Prompt dependency, large codebase and the limited contextual capacity.

Continued on next page

ID	AI Technique	RQ2: Target Issue	RQ1 and RQ3
[17]	LLM (GPT-4 and Claude 3 Opus) and static analysis tools (CodeQL and SpotBugs)	17 CWE including (89, 256, 476, 523, 549, 566, 606)	RQ1: Outperforms static analysis tools in terms of F1. RQ3: Monetary and time cost, CWE misclassification
[82]	LLM (GPT-4 and CodeLlama) and static analysis tool(CodeQL)	25 distinct CWE including (22, 79, 89, 125, 190, 476, 787)	RQ1: Mean 0.62 accuracy, 0.71 F1 across all datasets RQ3: , Hallucination, multi-file blindness, CWE misclassification
[60]	LLM + graph + static validation	Software vulnerabilities	RQ1: Improves F1 (+18.7%) and (+27.75%) average over baseline. RQ3: Lack clear explanation, needs verification and struggle to process complex code.
[81]	Multiple LLMs	Security vulnerabilities + logic errors + maintainability issues	RQ1: Comparable precision, lower recall; poor scalability. RQ3: False positives, inconsistent with larger codebases, hallucinations, prompt dependency.
[67]	LLM + static analysis	Error-handling bugs (return-value misuse in C)	RQ1: Higher recall (from an average of 52.55% to 77.83%) and higher F1-score (from an average of 0.612 to 0.804) than static analysis. RQ3: Nondeterminism, hallucinations.

Continued on next page

ID	AI Technique	RQ2: Target Issue	RQ1 and RQ3
[68]	ConSynergy framework (LLM semantics + static analysis structural precision)	Concurrency bugs (data races, deadlocks, atomicity violations)	RQ1: Average precision and recall of 80.0% and 87.1%, higher F1 but slower; handles non-compilable code. High precision scores across all models (e.g., 1.000 for GPT-4o and Claude 3.5 Sonnet on Juliet) RQ3: Recall decay, context limits.
[75]	LLM-generated static analyzers	10 kernel vulnerabilities (null pointer, buffer overflow, integer-overflow, use-after-free, etc.)	RQ1: Finds 92 new bugs beyond traditional tools. RQ3: Concurrency issues; multi-threaded code, locking schemes.
[77]	LLM semantic analysis	9 security smells including (hard-coded secrets, weak crypto, empty passwords, etc.)	RQ1: Higher precision/F1 than GLITCH, state-of-the-art security smell identification tool. RQ3: Prompt and validation dependency.
[61]	Graph learning + LLM	Security vulnerabilities (buffer overflow, integer overflow, uninitialized variables, double-free and use-after-free)	RQ1: High precision and recall; efficient training. 96.46% F1 score, 97.82% precision, and 95.14% recall RQ3: No line-level vulnerability localization

Continued on next page

ID	AI Technique	RQ2: Target Issue	RQ1 and RQ3
[78]	ML + Abstract Syntax Tree (AST)	Security vulnerabilities (CWE-119, CWE-120, CWE-469, CWE-476, CWE-20, CWE-457, CWE-805)	RQ1: Outperforms baseline models (Code2vec and Code2vec + MLP). RQ3: Localization and interpretation
[76]	ML algorithms (Random Forest, Extreme Boosting, Decision Tree etc)	Security vulnerabilities (complexity-related defects)	RQ1: High recall (more than 0.8) but high false positives. RQ3: Low interpretability.
[59]	DL (BiGRU + attention mechanisms)	Security vulnerabilities (zero-day + known flaws)	RQ1: +8.3% accuracy, +17.7% F1. RQ3: Limited generalizability to other programming languages, computational cost.
[64]	Random Forest + Non-Negative Matrix Factorisation	Smart contract vulnerabilities	RQ1: High performance (Acc 0.977, F1 0.808). RQ3: Semantic noise

Appendix B

Initially, generic prompts such as "What code smells are present in this code?" and "What security vulnerabilities exist in this code?" were used to evaluate the capabilities of the selected LLMs. However, the responses produced by these prompts were lacked the level of detail required for systematic benchmarking. Hence, more specific and structured prompts were developed.

Prompt for Blob

You are an expert Java software engineer specializing in code quality and software design patterns. Your task is to analyze the following Java class and determine whether it contains a BLOB (God Class) code smell. A BLOB (God Class) is defined as a class that:

- Has too many responsibilities and does too many things
- Contains an excessive number of methods and attributes
- Handles unrelated functionalities that should belong to other classes
- Is excessively large compared to other classes in the system

Step 1: Identify how many methods and attributes the class has.

Step 2: Identify how many distinct responsibilities the class handles.

Step 3: Determine if the class violates the Single Responsibility Principle.

Step 4: Based on your analysis, give your final verdict.

Answer strictly in this format:

VERDICT: YES or NO

REASON: (one sentence explaining why)

SEVERITY: Minor, Major, Critical

Prompt for Data Class

A DATA CLASS is defined as a class that:

- Only contains fields, getters, and setters

- Has no meaningful behavior or business logic

- Serves purely as a data container with no real methods

- Has all its data manipulated by other classes rather than itself

Step 1: Identify all methods in the class — are they only getters/setters/constructors?

Step 2: Check if the class contains any meaningful business logic or behavior.

Step 3: Determine if the class is merely a passive data holder.

Step 4: Based on your analysis, give your final verdict.

Answer strictly in this format:

VERDICT: YES or NO

REASON: (one sentence explaining why)

SEVERITY: Minor, Major, Critical

Prompt for Feature Envy

You are an expert Java software engineer specializing in code quality and software design patterns. Your task is to analyze the following Java method and determine whether it contains a FEATURE ENVY code smell.

A FEATURE ENVY method is defined as a method that:

- Uses data or methods from another class more than from its own class
- Makes excessive calls to getter methods of another object
- Seems more interested in another class than the one it belongs to
- Would be better placed in the class whose data it is using

Step 1: How many times the method accesses its own class's fields/methods.

Step 2: How many times the method accesses another class's fields/methods.

Step 3: Find if the method is more attached to another class than its own.

Step 4: Based on your analysis, give your final verdict.

Answer strictly in this format:

VERDICT: YES or NO

REASON: (one sentence explaining why)

SEVERITY: Minor, Major, Critical

Prompt for Long Method

You are an expert Java software engineer specializing in code quality and software design patterns. Your task is to analyze the following Java method and determine whether it contains a LONG METHOD code smell.

A LONG METHOD is defined as a method that:

- Is excessively long in terms of lines of code (typically more than 20-30 lines)

- Handles multiple tasks or responsibilities within a single method

- Contains deeply nested conditions or loops

- Is difficult to understand without reading every line carefully

- Should be decomposed into smaller, more focused methods

Step 1: Count the approximate number of lines of code in the method.

Step 2: Identify how many distinct tasks or operations the method performs.

Step 3: Check for deeply nested blocks, multiple loops, or complex conditionals.

Step 4: Based on your analysis, give your final verdict.

Answer strictly in this format:

VERDICT: YES or NO

REASON: (one sentence explaining why)

SEVERITY: Minor, Major, Critical

The detailed code smell prompts described above were used for the first sample of code smell category to provide the LLMs with explicit definitions and evaluation criteria. For all subsequent samples, a shorter prompt was employed to reduce redundancy and length of prompt.

Prompt for code smell

What code smells (Blob, Data Class, Feature Envy, or Long Method) are present in the attached code? Please use the definitions provided above when performing the analysis. Please answer in the above format.

Prompt for vulnerability detection

What security vulnerabilities in the attached java code? Give me the CWE name and also locate it. Give me a verdict of yes or no only.