**UNIVERSIDADE FEDERAL DE MINAS GERAIS**
**Instituto de Ciências Exatas**
**Programa de Pós-Graduação em Ciência da Computação**

Saymon da Silva Souza

# Automated Code Smell Detection with Large Language Models

Belo Horizonte
2025

Saymon da Silva Souza

# Automated Code Smell Detection with Large Language Models

**Final Version**

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Eduardo Magno Lages Figueiredo

Belo Horizonte
2025

2025, Saymon da Silva Souza.
Todos os direitos reservados

Souza, Saymon da Silva .

S729a Automated code smell detection with Large Language Models [recurso eletrônico] / Saymon da Silva Souza – 2025.

1 recurso online (82f. il., color.) : pdf.

Orientador: Eduardo Magno Lages Figueiredo.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 69-82.

1. Computação – Teses. 2. Engenharia de Software – Teses. 3. Detecção de anomalias (Computação) – Teses. 4. Revisão sistemática – Teses. 5. – Pesquisa Quantitativa – Teses. I. Figueiredo, Eduardo Magno Lages. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da computação. III. Título.

CDU 519.6*82(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz CRB 6/819 - Universidade Federal de Minas Gerais – ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# FOLHA DE APROVAÇÃO

# AUTOMATED CODE SMELL DETECTION WITH LARGE LANGUAGE MODELS

SAYMON DA SILVA SOUZA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores(a):

Prof. Eduardo Mango Lages Figueiredo - Orientador
Departamento de Ciência da Computação - UFMG

Prof. João Eduardo Montandon de Araújo Filho
Departamento de Ciência da Computação - UFMG

Profa. Juliana Alves Pereira
Departamento de Informática - PUC-Rio

Belo Horizonte, 03 de novembro de 2025.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Professor do Magistério Superior**, em 22/03/2026, às 18:36, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Juliana Alves Pereira**, **Usuária Externa**, em 23/03/2026, às 11:18, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **João Eduardo Montandon de Araujo Filho**, **Professor do Magistério Superior**, em 23/03/2026, às 13:04, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **4697588** e o código CRC **DA067A18**.

---

**Referência:** Processo nº 23072.267091/2025-80 SEI nº 4697588

# Acknowledgments

I would like to thank my family, friends, and professors. I would especially like to thank:

Jehovah, for granting me life and health, and for ensuring that nothing has ever been lacking throughout my life up to this moment.

My parents, Landerlei and Eunice, who have always fought to put me in the best situation possible and raise me in the best way they could, even in unfavorable situations, and who have always encouraged me to never stop learning.

My wife, Fernanda, who stood by my side through the most difficult moments, physical or emotional, never giving up on me and always encouraging me to pursue my dreams. You have been the best partner I could ever ask for, and this work would not have been possible without your unwavering support.

My mother-in-law, Lúcia, who has always helped me by dedicating her time, motivation, and care no matter what my wife and I needed.

My advisor, Prof. Eduardo Figueiredo, for his patience and dedication in teaching me how to be a researcher and for always pushing me to seek academic excellence.

My colleagues and friends at LabSoft, who always supported me and gave me good advice.

My boss at Claro, Marcio Cortez, who consistently supported me throughout my MsC program and created an environment that allowed me to attend classes and meet my job responsibilities without compromising my work.

My first boss, Prof. Arilo Dias, who gave me my initial professional opportunity and consistently encouraged me to bridge the gap between academia and the professional world.

The members of the defense committee of my dissertation, Prof. João Eduardo Montandon and Prof. Juliana Alves Pereira, for accepting the invitation to participate in this important step in my career and for providing valuable feedback on my work.

# Resumo

*Contexto:* Anomalias de código são indicadores de más práticas de projeto ou implementação que podem reduzir a qualidade e a manutenibilidade de sistemas de software. As ferramentas automatizadas de detecção existentes, como analisadores estáticos e modelos de aprendizado de máquina, utilizam métricas fixas e cobrem apenas um conjunto limitado de smells. Avanços recentes em Grandes Modelos de Linguagem (GMLs) abrem novas oportunidades para aprimorar a detecção de anomalias de código, mas seu papel ainda permanece pouco explorado. *Objetivo:* Esta dissertação investiga de que forma os GMLs podem contribuir para a detecção automatizada de anomalias de código. O trabalho tem como objetivo consolidar o conhecimento acerca das estratégias de detecção já existentes, além de criar um conjunto de dados que integre GMLs e ferramentas de análise estática. A partir dessa integração, realiza-se uma avaliação empírica para comparar a eficácia e o alinhamento dos GMLs em relação às ferramentas de análise estática e ao julgamento humano. *Metodologia:* Conduzimos três estudos complementares. Primeiro, realizamos uma Revisão Sistemática da Literatura (RSL) sobre estratégias de detecção automatizada de anomalias de código para JavaScript, identificando suas características e evolução. Em seguida, estendemos um conjunto de dados existente de sistemas Java com nove anomalias de código, incorporando saídas de quatro GMLs. A partir disso, construímos um conjunto-verdade automatizado por meio de votação majoritária. Por fim, realizamos uma avaliação empírica comparando GMLs, ferramentas de análise estática e julgamento humano, utilizando métricas de concordância (Kappa) e de eficácia (precisão, cobertura e F1-score). *Resultados:* Nossa RSL revelou que a maioria das estratégias de detecção em JavaScript se baseia em linters e foca em smells estruturais simples, enquanto smells mais complexos permanecem pouco explorados. Os resultados da avaliação empírica indicam que as ferramentas estáticas geralmente alcançam maior precisão, enquanto os GMLs obtêm maior cobertura. Os GMLs apresentaram bom desempenho em smells mais simples, como Long Method e Large Class, mas tanto as ferramentas quanto os GMLs tiveram dificuldades em detectar em smells mais complexos, como Feature Envy e Shotgun Surgery. Os GMLs complementam as ferramentas de análise, mas ainda não são soluções confiáveis de forma independente.

**Palavras-chave:** code smells; grandes modelos de linguagem; revisão sistemática da literatura; estudo empírico; estudo comparativo quantitativo.

# Abstract

*Context:* Code smells are indicators of poor design or implementation practices that can reduce the quality and maintainability of software systems. Existing automated detection strategies, such as static analyzers and machine learning models, only cover a limited set of smells and often rely on a set of fixed metrics. Recent advances in Large Language Models (LLMs) open new opportunities for improving code smell detection, but their role remains underexplored. *Objective:* This dissertation investigates how LLMs can support automated code smell detection. Specifically, it aims to consolidate knowledge about existing detection strategies, create a dataset that integrates LLMs and static analysis tools, and empirically evaluate the effectiveness and alignment of LLMs compared to tools and human judgment. *Method:* We conducted three complementary studies. First, we performed a Systematic Literature Review (SLR) of automated code smell detection strategies in JavaScript, identifying their features and evolution. Second, we extended an existing dataset of Java systems with nine annotated code smells, incorporating outputs from four LLMs (GPT-4o, Llama-3.3, Qwen2.5-Coder, and DeepSeek-R1, and built an automated ground truth through majority voting. Third, we carried out an empirical evaluation comparing LLMs, static analysis tools, and human judgment using agreement (Cohen's Kappa) and effectiveness metrics (precision, recall, and F1-score). *Results:* Our SLR results revealed that most current strategies rely on rule-based linting and focus on simple structural smells, while complex design-level smells remain underexplored. The empirical evaluation found that static analysis tools generally achieve higher precision, while LLMs achieve higher recall. GPT-4o and Llama-3.3 performed strongly for some structural smells, but both tools and LLMs struggled with design-level smells, such as Feature Envy and Shotgun Surgery. LLMs complement static analysis tools by improving coverage and alignment with developer perception, but they are not yet reliable standalone solutions.

**Keywords:** code smells; large language models; systematic literature review; empirical study; quantitative comparative study.

# List of Figures


1.1 Study Overview . . . . . . . . . . 15

2.1 Example of zero-shot prompting . . . . . . . . . . 22

3.1 Steps of the SLR . . . . . . . . . . 29
3.2 Frequency of Paper Publication by Year . . . . . . . . . . 34
3.3 Frequency of the Topic Discussion in Conferences and Journals . . . . . . . . . . 36
3.4 List of the Automated Strategies with Their Number of Primary Studies . . . 37

4.1 Steps of the Automated Ground Truth Creation . . . . . . . . . . 47
4.2 Prompt Used for Detecting Code Smells . . . . . . . . . . 49

5.1 Steps of the Empirical Evaluation . . . . . . . . . . 55

# List of Tables

2.1 Definitions of the Code Smells Used in this Dissertation . . . . . . . . . . . . . 21
2.2 Cohen's Kappa Benchmark Scale . . . . . . . . . . . . . . . . . . . . . . . . . . 23

3.1 Filtering Criteria . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2 Extracted Data From Each Automated Strategy and Its Description . . . . . . 33
3.3 Research Focus Areas and Paper Classification . . . . . . . . . . . . . . . . . . 35
3.4 Automated Code Smell Detection Strategies: Smells and Detection Technique 38
3.5 Detected Code Smells by Automated JavaScript Detection Strategies . . . . . 39
3.6 Automated Detection Strategies: Language, Availability and Features . . . . . 40
3.7 Evolution of Code Smell Detection over Time . . . . . . . . . . . . . . . . . . . 41

4.1 Systems in the Original Dataset . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.2 Static Analysis Tools Used to Detect Code Smells . . . . . . . . . . . . . . . . 46
4.3 Selected Large Language Models . . . . . . . . . . . . . . . . . . . . . . . . . 47
4.4 Code smells in the Automated Ground Truth . . . . . . . . . . . . . . . . . . . 50

5.1 Analysis of Agreement between Different Tools and LLMs . . . . . . . . . . . . 56
5.2 Effectiveness Metrics for the Automated Ground Truth . . . . . . . . . . . . . 59
5.3 Effectiveness Metrics for Human-based Ground Truth . . . . . . . . . . . . . . 60

# Contents

**1 Introduction** **12**
1.1 Problem Statement . . . . . 12
1.2 Research Method . . . . . 15
1.3 Results and Contributions . . . . . 16
1.4 Dissertation Outline . . . . . 17

**2 Background and Related Work** **19**
2.1 Code Smells . . . . . 19
2.2 LLMs and Prompt Engineering . . . . . 21
2.3 Accuracy and Agreement Metrics . . . . . 22
2.4 Related Work about Code Smell Detection . . . . . 23
2.5 Related Work about LLMs . . . . . 25
2.6 Chapter Summary . . . . . 26

**3 A Review of Automated Code Smell Detection: the JavaScript Case** **28**
3.1 Goal and Research Questions . . . . . 29
3.2 Search String and Criteria . . . . . 30
3.3 Databases and Data Extraction . . . . . 32
3.4 Landscape of Automated Code Smell Detection . . . . . 34
3.5 Features of Automated Detection Strategies . . . . . 37
3.6 Threats to Validity . . . . . 42
3.7 Chapter Summary . . . . . 42

**4 A Dataset of Automatically Detected Code Smells** **44**
4.1 The Original Dataset . . . . . 44
4.2 Automated Ground Truth Creation . . . . . 46
4.3 Used Prompt . . . . . 48
4.4 Voting Strategy of Tools and LLMs . . . . . 50
4.5 Threats to Validity . . . . . 51
4.6 Chapter Summary . . . . . 52

**5 Evaluating the Effectiveness of LLMs for Code Smell Detection** **53**
5.1 Goal and Research Questions . . . . . 54
5.2 Evaluation Steps . . . . . 54

5.3 Agreement between Code Smell Strategies . . . 55
5.4 Results with an Automated Ground Truth . . . 58
5.5 Effectiveness of LLMs against Human Judgment . . . 60
5.6 Threats to Validity . . . 62
5.7 Chapter Summary . . . 63

**6 Conclusion** **64**
6.1 Work Overview . . . 65
6.2 Main Contributions . . . 66
6.3 Study Implications . . . 67
6.4 Future Work . . . 68

**Bibliography** **69**

# Chapter 1

# Introduction

Code smells are symptoms of poor design or implementation choices that, although not actual defects, can gradually diminish the quality and maintainability of software systems [34]. They typically appear as patterns, such as large classes, overly long methods, or methods that rely excessively on external data [62]. When ignored, these smells accumulate as technical debt, making systems more difficult to understand, more fragile, and more expensive to evolve [57]. Therefore, identifying them early is a key activity in sustaining healthy codebases [1]. Although developers can detect smells through manual review, this process is subjective, error prone, and impractical for large or fast-evolving projects [68].

To mitigate this problem, a variety of automated detection strategies have been proposed [29], including rule-based static analyzers [87], metric-driven approaches [31], and machine learning models [10]. These strategies have proven useful, but they also have important shortcomings: they often disagree with each other [62, 94], do not necessarily reflect the opinion of developers about what constitutes a smell [131], and struggle with more subtle design issues that require contextual understanding [75]. Recent progress in Large Language Models (LLMs) introduces a new opportunity to address these gaps. Since LLMs are trained in large collections of code and natural language [26], they can capture both syntactic patterns and semantic relationships [46], allowing them to reason about code in ways that go beyond rigid thresholds or static rules [56]. This capability positions LLMs as promising allies to existing strategies, with the potential to improve alignment with human judgment and extend the range of smells that can be automatically detected.

## 1.1 Problem Statement

The literature on code smell detection has produced a variety of strategies, ranging from manual inspection [89, 112, 114] to automated strategies, such as rule-based static

analysis [22, 32, 87], metric-driven methods [28, 57, 113], and machine learning models [10, 23, 33]. Although these strategies have advanced the field, they remain fragmented and mostly focus on a few programming languages [29]. Furthermore, there is little empirical evidence that advanced machine learning techniques, such as Large Language Models (LLMs), can match or outperform these strategies tools in real-world projects. Finally, existing datasets are often limited in scope or focus on a few types of smells [62], making it difficult to evaluate new techniques fairly. Finally, LLMs have recently shown promise in software engineering tasks, such as code generation [17, 25, 59], repair [19], and code review [99], but their role in code smell detection remains underexplored [103].

One of the first challenges present in the literature is the absence of a consolidated view of automated detection strategies [42]. This is especially true for JavaScript. Although other programming ecosystems have begun adopting AI-based techniques for code smell detection, we are not aware of their use in the context of JavaScript. Furthermore, automated detection strategies differ in scope [58, 75], supported smells [14, 30], detection techniques [51, 116], and evaluation practices [4, 37], making it difficult for researchers and practitioners to compare them or select the most appropriate ones for their needs. Some automated strategies are industry-driven and actively maintained [111], while others are research prototypes that are no longer updated [28], further complicating adoption. Without a consolidated view of the field, it is challenging to identify research gaps, understand the evolution of detection strategies, or assess the extent to which modern strategies, such as AI-based techniques, have been adopted.

**Research Problem 1:** Lack of a consolidated view of automated detection strategies.

Closely related to the lack of synthesis is the problem of limited datasets for evaluating detection strategies. Existing datasets for code smell detection are often limited to a small set of systems. Moreover, many datasets focus on a narrow range of smells [16, 29, 62], such as Long Method or Large Class [42], while neglecting more complex design-level issues, such as Feature Envy or Shotgun Surgery. This imbalance makes it difficult to train or evaluate new detection techniques fairly, particularly those based on machine learning or LLMs, which require large and diverse datasets. Finally, most datasets are based exclusively on static analysis tools [94], which may not be enough to provide a reliable landscape of automated code smell detection. These challenges present a clear need for datasets that integrate multiple perspectives, including static analysis tools, LLMs and human evaluations, while also being extensible to new smells, languages, and systems.

**Research Problem 2:** Lack of datasets of automated code smell detection strategies including LLMs.

Building on these limitations, another open problem is the lack of understanding of how LLMs compare to other automated code smell detection strategies, such as traditional static analysis tools. Unlike static analysis tools, which rely on predefined rules or metrics, LLMs can capture semantic and contextual information from source code, potentially enabling them to detect smells that static analysis tools miss. However, it is unclear whether LLMs can match or surpass traditional strategies in real-world situations and for which types of smells they are most effective. The role of LLMs in code smell detection remains speculative, which limits their adoption in practice.

**Research Problem 3:** Limited understanding of how LLMs compare to static analysis tools in code smell detection.

Finally, even if LLMs prove effective compared to static analysis tools, a critical question remains about their alignment with human judgment. Automated strategies use different metrics that lead to differences in what each strategy considers a code smell. Similarly, LLMs can detect patterns that might not align with human intuition, potentially creating false positives detections or missing subtle design issues that developers would recognize. Since code smells are inherently subjective to some extent, it is essential to evaluate how well automated strategies align with human judgment. However, there is little empirical evidence to compare LLMs, static analysis tools, and human evaluations side by side. Without such evidence, it is difficult to determine whether LLMs can provide more developer-aligned results, or whether they simply replicate the inconsistencies of existing automated strategies. This uncertainty limits the practical usefulness of LLMs and prevents practitioners from confidently integrating them into their workflows.

**Research Problem 4:** Uncertainty about alignment of LLMs with human judgment in code smell detection.

## 1.2 Research Method

Figure 1.1 presents an overview of the studies developed for this dissertation, highlighting how each one addresses specific research problems. We proposed and executed three studies that complement each other in order to address the problems described in the previous section.

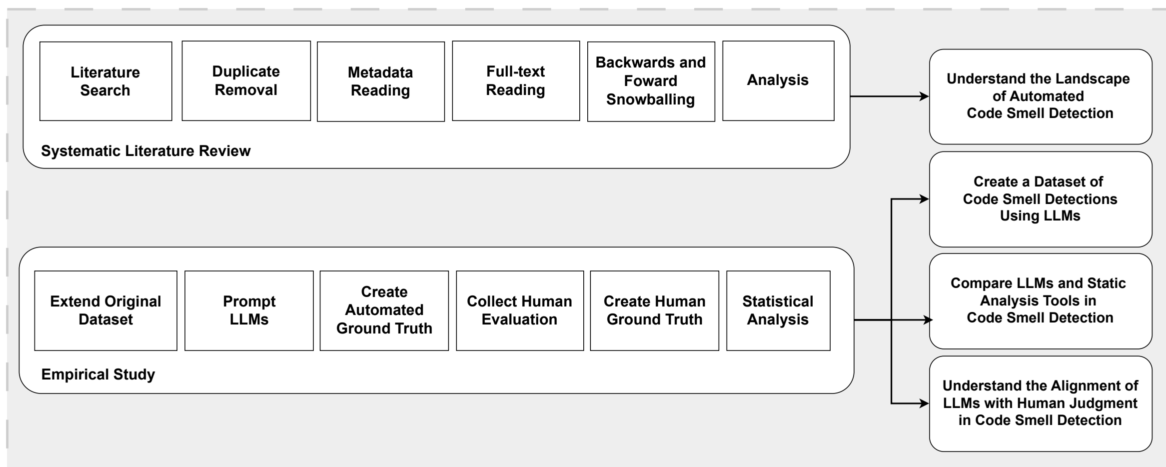



Figure 1.1: Study Overview

Source: Elaborated by the author.

**Study 1:** We conducted a Systematic Literature Review (SLR) to address the lack of a consolidated view of automated detection strategies, discussed in *Research Problem 1*. Following established guidelines in the literature [55], we performed eight steps: we defined research questions, selected databases, constructed a search string, applied it to multiple digital libraries, performed duplicate removal, performed a metadata read, performed a full-text read and performed a backward and forward snowballing. This process resulted in a curated set of primary studies, from which we extracted detailed information about each automated strategy, including supported smells, detection techniques, availability, and maintenance status.

**Study 2:** This study introduced a new dataset of automatically detected code smells to address *Research Problem 2*. We started by extending a well-established dataset of Java systems with nine types of code smells [94]. Subsequently, we randomly sampled a representative subset of classes from the original dataset. To incorporate LLMs into the new dataset, we selected four representative models, two proprietary and two open-source, based on their popularity, availability, and suitability for code-related tasks. We also modified a zero-shot prompt present in the literature [103] that clearly listed the nine target code smells and standardized the expected output format. Furthermore, we executed each LLM on the sampled instances and collected their outputs. Finally, we

built an automated ground truth through a majority voting strategy with the output of both static analysis tools and LLMs.

**Study 3:** The third study expands our empirical investigation. This study provides a comprehensive analysis on the role of LLMs in code smell detection created to address *Research Problem 3* and *Research Problem 4*. The evaluation followed a structured protocol with three main steps. First, we measured agreement between detectors using Cohen's Kappa to assess consistency between tools and LLMs. Second, we evaluated the effectiveness of all detectors by comparing them with the automated ground truth created in Study 2, calculating their precision, recall, and F1-score. Third, we evaluated the effectiveness of all detectors against a human-based ground truth, where 76 participants manually evaluated a sample of 268 instances, allowing us to determine how well each detector aligned with human perception.

## 1.3 Results and Contributions

Our SLR in the first study revealed that most automated detection strategies for JavaScript rely on rule-based linting and focus on simple structural smells, such as Long Method and Large Class, while more complex design-level smells remain underexplored. For practitioners, this means that widely used tools may provide useful feedback on basic structural issues, but they cannot be relied upon to capture deeper design flaws. For researchers, the review highlights the need to explore new approaches, including AI-driven techniques, to address these gaps. The review also showed that industry-driven tools dominate the ecosystem and continue to receive maintenance, while research-driven tools often lack long-term support. This result suggests that practitioners should prioritize industry-maintained tools for daily use, while researchers should focus on bridging the gap between academic prototypes and sustainable, real-world solutions. The results of this first study were submitted for publication in a Brazilian journal.

Among the results of our second study, we addressed the lack of robust datasets by extending an existing collection of Java systems [94] and constructing an automated ground truth that combined outputs from static analysis tools and LLMs. For practitioners, the automated code smell dataset provides a benchmark for comparing tools and models in realistic scenarios. Furthermore, it offers a reproducible and extensible foundation for researchers to evaluate new detection strategies, including the integration of new models and prompt techniques.

Our third study presented an empirical evaluation of LLMs compared to static analysis tools and human judgment. The agreement analysis showed that LLMs and

tools align moderately well for structural smells, yet poorly for those at design-level. The effectiveness analysis against the automated ground truth revealed that static analysis tools generally achieve higher precision, while LLMs achieve higher recall, with GPT-4o standing out for Data Class and all detectors performing well for Long Method. Finally, the evaluation against a human-based ground truth demonstrated that LLMs can surpass static analysis tools for structural smells, although tools remain more reliable for coupling-related and design-level smells. Together, these results show that LLMs are promising complements to existing tools, broadening coverage of structural smells. However, they are not yet reliable standalone solutions for complex design issues. Our main contributions from this work include:

1. A catalog of automated code smell detection strategies for JavaScript, each characterized by availability, smells detected, among other features;
2. An automated ground truth of code smell detections created through majority voting between four static analysis tools and four LLMs;
3. An empirical evaluation of LLMs compared to static analysis tools, analyzing agreement, precision, recall, and F1-score in nine code smells;
4. A human-based ground truth and comparative study that assesses how well LLMs and tools align with developer perception of code smells;
5. Replication packages for Study 1 [1], Study 2 [2] and Study 3 [2];

## 1.4 Dissertation Outline

The remainder of this dissertation is structured as follows.

Chapter 2 introduces important concepts that are used throughout this dissertation, such as code smells, LLMs and prompt engineering, and accuracy metrics. We also discuss some related work with respect to relevant topics.

Chapter 3 presents our systematic literature review, describing the steps followed, the digital databases and search string used, and so forth. It also discusses the respective findings obtained from our investigation of the literature regarding automated code smell detection strategies for JavaScript systems.

[1] https://zenodo.org/records/15757269
[2] https://zenodo.org/records/16790193

Chapter 4 details the initial dataset that served as the foundation for our LLM studies. This chapter discusses the process of constructing the automated ground truth, including the selection of LLMs and the design of the prompts used for code smell detection.

Chapter 5 describes the empirical study on the evaluation of how different LLMs perform in detecting code smells, comparing their performance against static analysis tools. It also outlines the study protocol, including our objectives and research questions, and presents a thorough discussion of the results.

Chapter 6 concludes this dissertation by summarizing its main contributions and findings, and by offering suggestions for future research.

# Chapter 2

# Background and Related Work

Code smells are indicators of potential problems in code that may not be bugs themselves, but suggest deeper design issues [34]. Recognizing these smells early allows developers to address underlying problems before they escalate, making code easier to understand, maintain, and extend [1]. By systematically identifying and refactoring code smells, teams can improve software quality, reduce technical debt, and foster a more sustainable development process [66]. Code smells can be detected in the source code by using manual or automated analyses [68, 89, 112, 114]. Automated strategies usually rely on different detection approaches, such as metric-based [28, 57, 113] and machine-learning techniques [7, 10, 23, 54], such as LLMs. In fact, some preliminary studies have investigated the use of LLM to detect and refactoring code smells [103]. In this chapter, we present concepts that are discussed throughout this dissertation.

The chapter is structured as follows. Section 2.1 describes the code smells that are used in this dissertation, highlights their practical significance, and examines the recurring challenges in their detection. Section 2.2 presents LLMs, their uses in Software Engineering and the most used prompt engineering techniques. Section 2.3 discusses the metrics that are used in this dissertation. Section 2.4 presents related work about existing literature reviews on automated code smell detection strategies, while Section 2.5 discusses related work on the use of LLMs in Software Engineering tasks. Finally, Section 2.6 concludes this chapter and introduces the next one.

## 2.1 Code Smells

Code smells are symptoms or indicators of potential code quality degradation that can affect software maintainability and evolution [34]. They not only impact on code understandability, reusability, and extensibility [104, 130], but they may also be the source of bugs and code instability [43, 79, 94, 95]. Refactoring is an activity in which the code is modified to improve its internal quality without changing its external behavior [34].

Therefore, there is a direct relationship between refactoring and code smells. Due to the potential of this type of problem to compromise maintenance effort, cost, and software quality, the subject has been thoroughly discussed in the literature [10, 27, 29, 89, 96, 112, 113].

Due to time constraints or community-related factors, developers often lack the time or motivation to control the complexity of the system and design effective solutions before implementing their changes [106]. However, even when there is motivation, some code smells are difficult to recognize [57, 60]. Therefore, they pose a significant threat to the cost and maintainability of software systems. In fact, previous research [1] has shown that code smells significantly hinder the ability of developers to understand source code and make affected classes more susceptible to changes and errors. Although the term code smell originally applied to code that breaks the principles of object-oriented programming, it soon expanded to include more languages and programming paradigms [96]. A study by Marinescu [66] shows that files with code smells are more likely to have maintenance problems, highlighting the need to detect and fix these issues to improve software quality. Manual analysis can produce satisfactory results [63, 112], but this can require significant time and effort from the development team. Therefore, more and more automated strategies have emerged to reduce the manual effort needed to prevent and correct code smells [31, 117, 120].

Several techniques to identify code smells have been proposed in the literature, such as manual code inspection [63, 112], static analysis tools [28, 37, 87, 113], refactoring opportunities [31], change history analysis [78], and machine learning models [20, 23, 63, 95]. Although static analysis tools, such as linters, are widely used by developers to detect code smells, several studies in the literature indicate they have poor agreement with the perception of what developers agree is a code smell [29, 131]. Previous work [20, 23, 72] also indicates the low accuracy of some classic machine learning models, such as Naive Bayes, Decision Tree, and Random Forest, to detect code smells. More importantly, although some preliminary studies have been recently published in this topic [128], we still lack strong empirical evidence on the effectiveness of LLMs to support code smell detection [103].

Table 2.1 describes the 9 code smells used in this dissertation. The first column shows the smell name, while the second column briefly defines the code smells. More details on their definition can be found in the books of Fowler [34], and Lanza and Marinescu [57]. Large Class, Data Class, and Refused Bequest exist at the class level. Meanwhile, Feature Envy, Intensive Coupling, Dispersed Coupling, Long Parameter List, Shotgun Surgery, and Long Method are present at the method level. We chose these particular code smells because they are well supported by automated detection strategies [87, 113] and cover a wide range of source code issues.

Table 2.1: Definitions of the Code Smells Used in this Dissertation

| Code Smell | Definition |
|---|---|
| Data Class | A class composed mainly of fields and getters/setters, with little or no meaningful behavior. [34] |
| Dispersed Coupling | A method that depends on many other classes but with low coupling intensity. [57] |
| Feature Envy | A method that accesses members of other classes more than its own. [34] |
| Intensive Coupling | A method that heavily interacts with one or a few other classes, forming a tight cluster. [57] |
| Long Method | A method that is excessively long or takes on too many responsibilities. [34] |
| Large Class | A class that handles multiple responsibilities or contains many lines of code. [34] |
| Long Parameter List | A method signature that requires an excessive number of parameters. [34] |
| Refused Bequest | A subclass that overrides or ignores most inherited behavior, indicating a poor inheritance fit. [34] |
| Shotgun Surgery | A change in one module forces many small changes scattered across other modules. [34] |

## 2.2 LLMs and Prompt Engineering

In language processing, traditional Language Models (LMs) have long served as the foundation for text generation and understanding [69]. Advances in computational power, machine learning, and access to large datasets have driven a major shift toward LLMs [92, 135]. Trained in massive and diverse corpora, LLMs have shown remarkable abilities to mimic human language [133], driving significant changes across many fields in Software Engineering [5, 6, 17, 18, 56, 97, 99, 121, 136]. Their capacity to learn from large-scale data and produce coherent, human-like text is narrowing the gap between machine and human communication [36]. As a result, LLMs have become powerful tools for researchers and developers, becoming an important part in a transformative era for language processing and related disciplines.

Prompt engineering means creating clear, specific instructions, called prompts, to guide what a model says or does, without changing how the model itself works [46]. By carefully writing these prompts, it is possible for the model to do many different tasks well, just by changing the instructions [125]. This is different from older methods, where you had to retrain or adjust the model for each new job [69]. Prompt engineering makes LLMs more flexible and powerful, opening up new ways to use these models in many areas. Prompts are usually made up of instructions, questions, input data, and sometimes examples [88]. To obtain the desired answer from a model, the prompt should include at least instructions or a question [73]. One prompt engineering technique that is commonly applied in Software Engineering is Zero-shot prompting [46].

**Zero-shot prompting** uses generic prompts to tell the model what to do without presenting training examples [123]. Figure 2.1 presents an example of prompt using this technique. The model gets a description of the task in the prompt, but it does not see any examples with correct answers. Instead, it uses what the model already knows to try to solve the new task based on the instructions it receives.

Prompt

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

Figure 2.1: Example of zero-shot prompting

Source: Elaborated by the author based on the work of Wei et al. [125].

## 2.3 Accuracy and Agreement Metrics

Accuracy metrics are statistical measures used to evaluate the performance of a predictive model or tool [105]. These metrics are used to assess how well the model or tool is able to make correct predictions on a given set of data. For instance, in the context of comparing LLMs with static analysis tools, they are used to evaluate the instances obtained by both automated code smell detection strategies. Such metrics are essential because they provide a standardized way to compare different approaches, regardless of the domain. We should highlight that no single metric is sufficient to capture all aspects of effectiveness and that a combination of complementary measures (e.g., precision, recall, and F1-measure) is necessary to obtain a balanced evaluation. Moreover, these metrics help to identify trade-offs between detecting as many relevant instances as possible and avoiding false detections, which is directly applicable when assessing the reliability of code smell detection.

Precision [105] measures the proportion of true positives (correctly predicted positive outcomes) out of all positive predictions made by a model or tool. With respect to the output of LLMs and static analysis tools, precision is the ratio between the number of true positive detected smells and the total number of detected instances. This metric measures how often the instances detected by LLMs and static analysis tools are correct. Recall [105] measures the proportion of true positives (correctly predicted positive outcomes) out of all actual positive outcomes in the dataset. In the context of code smell detection, recall is the ratio between the number of true positive detected code smells and

the number of existing code smells in a dataset. This metric measures how complete the output of the LLM or static analysis tool is in terms of the existing code smells in the dataset. F1-Measure [105] is a weighted average of precision and recall. This metric provides a single score that balances both precision and recall to provide an overall measure of the effectiveness of a model. It can also be obtained by calculating the harmonic mean of precision and recall.

Lastly, an agreement metric is a statistical measure that is used to evaluate the level of agreement between two or more raters or judges who evaluate a set of data [41]. Agreement is an important tool for assessing the reliability and validity of data collected by multiple raters or judges. In the context of the comparing LLMs and static analysis tools for code smell detection, this metric measures the agreement between multiple automated strategies when detecting code smell instances from the same dataset. There are several types of agreement metrics that can be used, depending on the nature of the data and the type of analysis being performed. Some commonly used agreement metrics include Cohen's Kappa, Fleiss's Kappa and the Intraclass Correlation Coefficient [41]. In this work, we chose Cohen's Kappa coefficient [41]. Table 2.2 summarizes how we interpret the strength of agreement between two raters using Cohen's Kappa. The values range from -1 (complete disagreement) to 1 (perfect agreement), with 0 indicating that the agreement is not better than chance. Unlike simple percentage agreement, Cohen's Kappa is a statistical measure that accounts for the likelihood of agreement occurring by chance, providing a more reliable and nuanced evaluation.

Table 2.2: Cohen's Kappa Benchmark Scale

| **Kappa Statistic** | **Strength of Agreement** |
| --- | --- |
| $< 0.20$ | Poor |
| 0.21 to 0.40 | Fair |
| 0.41 to 0.60 | Moderate |
| 0.61 to 0.80 | Good |
| 0.81 to 1.00 | Very Good |

## 2.4 Related Work about Code Smell Detection

To our knowledge, no prior work has conducted a systematic literature review and a thorough evaluation of automated code smell detection strategies for JavaScript software systems, as we present in Chapter 3. However, we identified some related studies. Recent empirical studies have explored how code smells relate to software maintainabil-

ity, considering both system-level indicators and their direct impacts on maintenance effort [104, 130]. Yamashita and Counsell [130] found that code smells can highlight areas that need maintenance, but their usefulness is limited when comparing systems of different sizes; Expert judgment remains the most adaptable approach. Sjøberg et al. [104] showed that after adjusting for file size and change frequency, code smells were not significantly related to increased maintenance effort. In general, these studies suggest that while code smells have some value, prioritizing code size and change management is more effective in reducing maintenance effort.

**Empirical Studies on Code Smell Detection:** Recent years have seen notable progress in understanding code smells, with research covering mapping studies, detection techniques, and empirical analyses of their impact [4, 11, 28, 29, 32, 50]. For instance, Barros and Adachi [11] mapped 26 types of code smells in JavaScript, noting that most are adaptations from other languages and that research in this area is still limited. Fard and Mesbah [28] introduced JSNose, an automated strategy using static and dynamic analysis to detect 13 code smells, finding that issues such as, Lazy Object, Long Method, Closure Smells and excessive global variables, are especially common. Almashfi and Lu [4] also developed an automated detection strategy, which reinforced the need for automated solutions. Johannes et al. [50] conducted a large-scale study on 15 JavaScript projects, showing that files with code smells have at least a 33% higher risk of faults, with "Variable Re-assign", "Assignment in Conditional Statements", and "Complex Code" being particularly persistent and fault-prone.

Fontana et al. [32] conducted a literature survey covering seven automated code smell detection strategies: Checkstyle, DÉCOR, inFusion, iPlasma, JDeodorant, PMD, and Stench Blossom. In addition, they experimentally assessed four of these strategies, Checkstyle, inFusion, JDeodorant, and PMD, applying them to six different versions of the same software system. Their results showed that the automated strategies produced notably different detection outcomes for the same code smell, with some overlaps among the findings. Regarding strategy agreement, they observed significant concordance only in the detection of two code smells: Large Class and Long Parameter List.

Fernandes et al. [29] conducted a systematic literature review and a comparative analysis of code smell detection tools. They surveyed 84 automated strategies reported in the literature, of which 29 are available online, and classified them according to supported programming languages, detection approaches, and targeted smells. Furthermore, they empirically evaluated four automated strategies, inFusion, JDeodorant, PMD, and JSpIRIT, using two Java software systems as input and focusing on the detection of Large Class and Long Method smells. Their results show a high degree of agreement among strategies, with notable redundancy and variability in detection outcomes, as well as some usability issues. They concluded that while many automated strategies exist, challenges remain in tool effectiveness and user experience.

**Human Judgment vs. Automated Detection in Identifying Code Smells:** The debate between human judgment and automated detection in the identification of code smells has been actively explored in the literature, with studies examining how developer expertise and automated strategies can complement each other [47, 93, 98]. In this context, Saboury et al. [93] found that although developers consider detecting God Class easy, their agreement is low, while automated and metric-based methods align surprisingly well with human assessments. Hozano et al. [47] showed that developers rarely agree on code smell identification, mainly due to individual experience, emphasizing the value of automated methods to reduce subjective bias. Schumacher et al. [98] demonstrated that metric-based algorithms effectively support detection and, when combined with manual review, increase confidence and reduce inspection effort. In general, these studies suggest that the integration of automated strategies with human judgment can improve consistency, accuracy, and efficiency in code smell detection. Unlike previous work, this dissertation provides an overview of the state of the art in automated code smell detection strategies for JavaScript through an SLR (Chapter 3). Furthermore, this review outlines the key features of each strategy, revealing their advantages and limitations.

## 2.5 Related Work about LLMs

LLMs have demonstrated significant promise in Software Engineering and are employed across a wide variety of tasks. These tasks include code generation [17], software migration [5, 121, 136], program repair [56], code review [99], and test case generation [6, 18, 97]. Their wide-ranging use in Software Engineering comes from the fact that they are trained in massive collections of code and natural language, which strengthens their ability to process human language as well as to interpret and understand source code. In this section, we detail studies that specifically address the use of LLMs for some software engineering tasks.

**Recent Advances in Code Generation and Evaluation using LLMs** Recent research has examined both the capabilities and limitations of LLMs in code generation and broader software engineering activities [2, 17, 25, 59, 73]. Liu et al.[59] conducted a systematic evaluation of ChatGPT's code generation, shedding light on its strengths and persistent weaknesses. Dong et al.[25] introduced a self-collaboration approach in which multiple specialized LLM agents work together to enhance code generation, achieving significant improvements over single-agent setups. Similarly, Caumartin et al.[17] demonstrated that open-source Llama models, when properly tuned, can reach performance levels comparable to ChatGPT in code refinement tasks. O'Brien et al.[73] observed that

prompt engineering using TODO comments can either support or affect Copilot's handling of technical debt depending on the clarity of the comments. Finally, Al Madi [2] reported that Copilot's generated code is generally as readable as human-written code, although developers tend to review it less carefully, raising automation bias issues.

**LLMs for Code Smell Detection.** Recent research has explored the use of LLMs to detect and address code smells in various contexts [49, 103, 128]. Silva et al. [103] evaluated ChatGPT's ability to identify four traditional code smells, finding that listing specific smells in the prompt improved overall performance, although challenges remained for more complex smells. Wu et al. [128] proposed iSMELL, an ensemble approach that combines LLMs with automated code analysis tools, which outperformed both standalone LLMs and expert tools to detect and fix certain code smells. Jiang et al. [49] focused on gas-wasting code smells in Ethereum smart contracts, using GPT-4 to identify and help fix inefficiencies, resulting in significant cost savings. Compared to these works, our study presented in Chapter 5 investigates a broader set of Java code smells, evaluates both proprietary and open-source LLMs, and provides datasets with both automated and human-based ground truths.

**Small Language Models for Code Smell Detection.** Recent work has investigated the feasibility of using Small Language Models (SLMs), defined as language models with fewer than 8 billion parameters, for the classification of code smells in Python systems [76]. Oliveira et al. [76] evaluated SLMs in the detection of two common code smells, Long Method and Long Parameter List, comparing their performance with traditional static analysis tools, as well as with machine learning and deep learning models. Their study found that SLMs, particularly when guided by chain-of-thought prompting, achieved competitive F1-scores (up to 0.87 for Long Method), offering a promising balance between accuracy, interpretability, and deployment efficiency in resource-constrained environments. This study highlights the potential of lightweight language models for explainable and practical code smell detection, especially in environments where the use of larger models is not feasible. In contrast to this previous study, this dissertation investigates large language models with parameter counts starting at 32.5B parameters.

## 2.6 Chapter Summary

In this chapter, we introduced the fundamentals of code smells, highlighting their impact on software maintainability and the inherent difficulties in their detection. We then outlined the core concepts of LLMs and reviewed the most common prompt engineering strategies reported in the literature. Furthermore, we also discussed key evaluation met-

rics, including precision, recall, F1-Measure, and agreement. Finally, we examined recent studies that are taking advantage of LLMs for code smell detection, emphasizing their potential to capture code context and patterns more effectively. Although LLMs demonstrate considerable promise, challenges related to data quality, evaluation consistency, and practical integration still need to be addressed.

In the next chapter, we describe our Systematic Literature Review (SLR) on automated code smell detection strategies. We describe our goal and research questions, detail the study design by presenting the search string and used filtering criteria, the chosen databases, and the data extraction process. Finally, we discuss the landscape of automated code smell detection strategies and their features.

# Chapter 3

# A Review of Automated Code Smell Detection: the JavaScript Case

Code smells can be found in the source code by manual checks or automated analysis [68]. Although code smells can be identified manually [112], this process becomes very time-consuming in large systems. To address this, several automated detection strategies have been developed to support developers in improving code quality [68, 80, 101], each using different types of information. Some strategies are software metrics [113], textual analysis [81], AST analysis [43], visualization support [70], and machine learning algorithms. Several studies relying on machine learning algorithms to detect code smells have been proposed in the literature [7, 54, 63, 65, 109]. Because many automated strategies have been proposed in the literature, it is difficult to list them all and specify which code smells they can detect [29].

Despite the popularity of JavaScript and the growing number of automated code smell detection strategies for this language, the current body of research remains fragmented and methodologically inconsistent [11]. This suggests an uneven coverage of code quality concerns, which may lead to tool misalignment with real-world development challenges [23]. Furthermore, while other programming ecosystems have begun adopting AI-based techniques for code smell detection, we are not aware of their use in the context of JavaScript [128]. Combined with the lack of benchmark datasets and standardized evaluation practices, this scenario complicates both empirical validation and tool evolution [100]. To fill these gaps in the literature, this chapter presents a Systematic Literature Review (SLR) [55] that not only catalogs existing automated detection strategies, but also identifies research gaps to the adoption of AI techniques, such as LLMs.

The remainder of this chapter is organized as follows. Section 3.1 presents the study goal and research questions. Section 3.2 presents the search string and details the filtering criteria. Section 3.3 presents the data extraction steps. Section 3.4 describes what we learned about the publication landscape of automated code smell detection strategies since JavaScript was released. Section 3.5 presents the features of automated detection strategies. Section 3.6 discusses some threats to the study validity. Finally, Section 3.7 concludes the chapter and introduces the next one.

## 3.1 Goal and Research Questions

Figure 3.1 outlines the process we followed to perform our SLR to support automated code smell detection. We rely on the Goal-Question-Metric template [12] to formally define our study goal (Step 1 of Figure 3.1) as follows: *analyze* the state-of-the-art of automated code smell detection strategies for JavaScript systems; *for the purpose of* (1) understanding the evolution and focus of research in this domain, and (2) building a comprehensive catalog and comparison of existing detection strategies based on their features; *with respect to* aspects such as the types of code smells detected, underlying analysis techniques, supported environments, and evaluation practices; *from the point of view of* software engineering researchers, tool builders, and practitioners; *in the context of* of peer-reviewed academic publications since 1999, the year that the term code smell became popular [34].

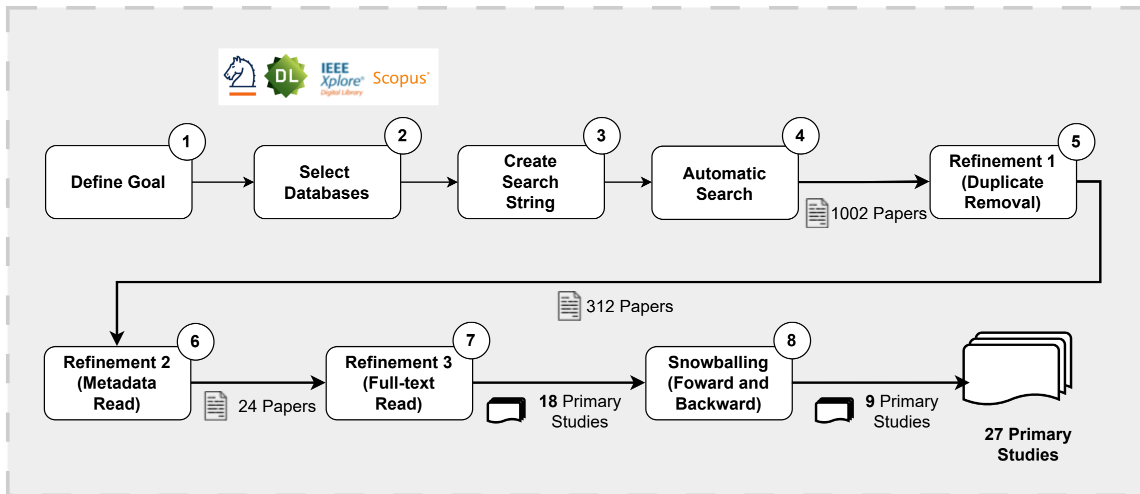



Figure 3.1: Steps of the SLR

Source: Elaborated by the author.

To achieve our study goal, we opted for a SLR based on strict literature guidelines [55, 126]. The SLR process involves three main stages: planning, conducting, and reporting. In the planning stage, the need for the review is identified, research questions are formulated, and a review protocol is established. The conducting stage involves executing this protocol, from selecting papers to extracting and analyzing data. The final stage is reporting the findings to the intended audience. Additionally, we defined the research questions ($RQ_s$) below aimed to guide our study directions.

**$RQ_1$:** *What is the publication landscape of automated code smell detection strategies in JavaScript since the term Code Smell became popular?* - To the best of our knowledge, there is no prior study that systematically summarizes existing automated

code smell detection strategies for JavaScript in a way that supports developers and researchers in selecting automated strategies that align with their specific needs. Our first objective is to understand the state-of-the-art in the research community with respect to the development and evaluation of such automated strategies. $RQ_1$ aims to fill this gap. To address it, we analyze metrics such as the number of publications over time, publication venues (e.g., conferences, journals), study types (e.g., tool proposals, empirical validations), and the timeline of the proposed strategies releases. These metrics provide a comprehensive view of the maturity, evolution, and research trends of the field.

**$RQ_2$:** *What automated code smell detection strategies were published so far and what are their main features?* - As far as we know, there has not been a comprehensive study that clearly identifies the main characteristics which demonstrate the strengths and weaknesses of current automated JavaScript code smell detection strategies. Furthermore, we have not found any research that provides recommendations on how these characteristics could guide the design of future automated strategies. $RQ_2$ addresses this gap by systematically extracting and classifying automated strategies characteristics, including detection approaches (e.g., rule-based, dynamic, AI-driven), types of code smells detected, interface modalities (e.g., CLI, GUI, IDE plug-ins), and the availability of automated strategies. These metrics enable a comparative analysis of automated approaches and help identify both existing limitations and opportunities for future research and strategy improvements.

## 3.2 Search String and Criteria

We created the search string (Step 3 of Figure 3.1) by considering various ways to identify *code smell* expressions, as well as references to *JavaScript*. We designed our search string to maximize the discovery of automated strategies. To construct it, we used the keywords *code smell*, *bad smell*, and *JavaScript*, incorporating the * wildcard to capture common word variations and suffixes.

*(( code smell* OR bad smell* OR code anomal* OR architectural smell* OR architectural anomal* OR design smell* OR design anomal* OR test smell* OR test anomal*) AND ( JavaScript OR Java Script OR ECMAScript OR ECMA Script OR ES6 ))*

We refined our search string through several rounds to improve the quality of the collected results. The initial version included specific tools and detection-related terms, which yielded a low number of results. By gradually removing these restrictive terms

and focusing on broader code smell and JavaScript keywords, the number of results increased. We briefly experimented with including terms like "code clone" and "duplicated code", which raised the results but shifted the focus too far from our study goal. Ultimately, incorporating terms related to test smells and keeping all relevant code smell and JavaScript synonyms, we arrived at a balanced, final search string without including excessive unrelated terms. We query primary studies in four electronic data sources (Step 2 of Figure 3.1): Scopus [1], ACM Digital Library [2], IEEE Xplore [3], and Springer [4]. We ran our final search string on May 31, 2025. We then imported BibTeX and text files, converted to BibTeX when needed, into the JabRef [5] tool to organize references.

We applied the same search string across all databases, adapting its format as needed to match the search requirements of each database. For instance, in IEEE Xplore, we divided the string into individual search terms to comply with its advanced search interface. Whenever possible, we also performed a preliminary filtering process by applying filters during the search, such as restricting publication years to exclude papers published before 1999, in line with our exclusion criteria. These adjustments helped ensure consistency and relevance in the studies we recovered from each source.

Our initial search yielded a total of 1,002 studies (Step 4 of Figure 3.1), with 220 papers collected from the ACM Digital Library, 104 from IEEE Xplore, 515 from Scopus, and 163 from Springer. We found duplicate and irrelevant results, which led to a refinement process. We applied a four-step refinement process to determine the final set of primary studies. Steps 5 to 8 of Figure 3.1 describe these steps. Three researchers participated in the process. In Refinement 1 (Step 5), we removed duplicates and non-academic works, such as technical reports and conference calls, reducing the initial set from 1002 to 312 studies. We then applied filtering criteria to remove irrelevant results.

Table 3.1 shows the filtering criteria, including four inclusion criteria and three exclusion criteria. For instance, we excluded studies published before 1999 since it was the year the term code smell was popularized [34]. As an example of inclusion criteria, we only included studies that used or proposed an automated detection strategy. For this study, we considered any software system specified by its authors as an automated strategy. In Refinement 2 (Step 6), we reviewed the metadata, applying the inclusion and exclusion criteria described in Table 3.1, which narrowed the selection to 24 studies. Refinement 3 (Step 7) involved a full-text review of these papers, eliminating those that did not propose or use an automated code smell detection strategy, leaving 18 primary studies. We relied solely on manual reading and did not use any software tools during the paper review process.

[1] https://www.scopus.com/home.uri
[2] http://dl.acm.org/
[3] http://ieeexplore.ieee.org/
[4] https://www.springer.com/
[5] http://jabref.sourceforge.net/

Table 3.1: Filtering Criteria

| Type | Criteria |
|---|---|
| Inclusion | It proposes or mentions an automated code smell detection strategy, an automated linting strategy and/or an automated strategy that fixes defects<br>It is focused on JavaScript language<br>It was written in English<br>It was published in Computer Science related venues |
| Exclusion | It is not a primary study<br>It was published before 1999<br>It is not a paper (conference, symposium or workshop), journal article or magazine article<br>It contains fewer than two pages |

With the initial set of primary studies set, we went to perform both forward and backward snowballing processes [55, 126] in these 18 primary studies (Step 8) using the same filtering criteria presented in Table 3.1, identifying 9 additional relevant studies. For backward snowballing, we collected the references cited by the primary studies, applied our inclusion and exclusion criteria, and then conducted a full-text review of the papers that met these criteria. For forward snowballing, we identified papers that cited our primary studies, again applied the inclusion and exclusion criteria, and performed a full-text read on the included papers.

All refinement steps followed a voting system, a process in which a paper could only advance to the next refinement step if it received two positive inclusion votes out of three researchers. This process resulted in 27 selected papers for data extraction based on our inclusion and exclusion criteria. When prioritizing references, we focused on papers proposing automated detection strategies; if unavailable, we relied on those using or citing such automated strategies. To supplement our findings, we searched for additional information about the automated strategies in online sources, including Google, GitHub, and other relevant repositories.

## 3.3 Databases and Data Extraction

We conducted a detailed full-text review of the 27 primary studies selected for data extraction. This process resulted in the finding of 22 automated code smell detection strategies focused on JavaScript. Regarding the publication year of each automated strategy, we followed a prioritized approach to determine the most accurate value. First,

we considered the year when researchers integrated a smell detection approach or algorithm into an existing automated strategy. If the integration year was unavailable, we used the publication year of the paper that introduced the automated strategy. In the absence of both, we adopted the publication year of the first commercial off-the-shelf version. Lastly, if none of the previous data points was available, we referred to the earliest documented release year found on the Web.

Table 3.2 presents the data extracted from the 22 automated strategies collected by executing the SLR protocol. We organized the extracted data for each automated strategy into a spreadsheet to support the analysis, along with additional relevant information from the reviewed papers that could assist in writing this study. During data extraction, we encountered some challenges, particularly with incomplete or missing automated strategy descriptions. For example, several papers did not clearly list the types of code smells detected or lacked a dedicated section outlining automated strategies features, such as supported languages or detection techniques. To address these gaps, we conducted supplementary Web searches in an effort to gather as much information as possible about the 22 automated strategies. The data extraction process and the validation of the information collected was performed in pairs, maintaining double validation.

Table 3.2: Extracted Data From Each Automated Strategy and Its Description

| **Name** | **Description** |
|---|---|
| Detected Code Smells | What code smells does the automated strategy detect? |
| Detection Technique | What is the problem modeling approach (Rule-based linting, Dynamic analysis, ...)? |
| Free for Use | Is it free for use? |
| Guide | Does it have a guide? |
| GUI | Does it have a GUI interface? |
| Industry or Research (IoR) | Was it proposed by the industry or researchers? |
| Language Developed | In what language was the respective automated strategy developed? |
| Plug-in | Is the respective automated strategy available as a plug-in? |
| Release Year | When did developers or researchers published the automated strategy? |
| Still Maintained | Did the corresponding automated strategy receive new updates in the last three years? |
| Automated Strategy Name | What is the automated strategy name? |

## 3.4 Landscape of Automated Code Smell Detection

In this section, we present the results obtained after briefly studying the automated detection strategies found. Figure 3.2 shows the frequency of publications on the subject per year. Notably, since 2017, the number of publications has increased, peaking in 2022 with six papers published. The rise in publications might be due to several factors. First, JavaScript is now used beyond client-side scripting, especially with Node.js, leading to larger and more complex projects that require better maintenance [71, 91]. Additionally, modern frameworks such as React, Angular, and Vue.js introduced new coding patterns, increasing the need for research on code smells [30]. Lastly, collaboration between academia and industry has also driven interest in automated detection. Advances in static analysis, machine learning, and quality assurance have made detection easier and more effective [23].

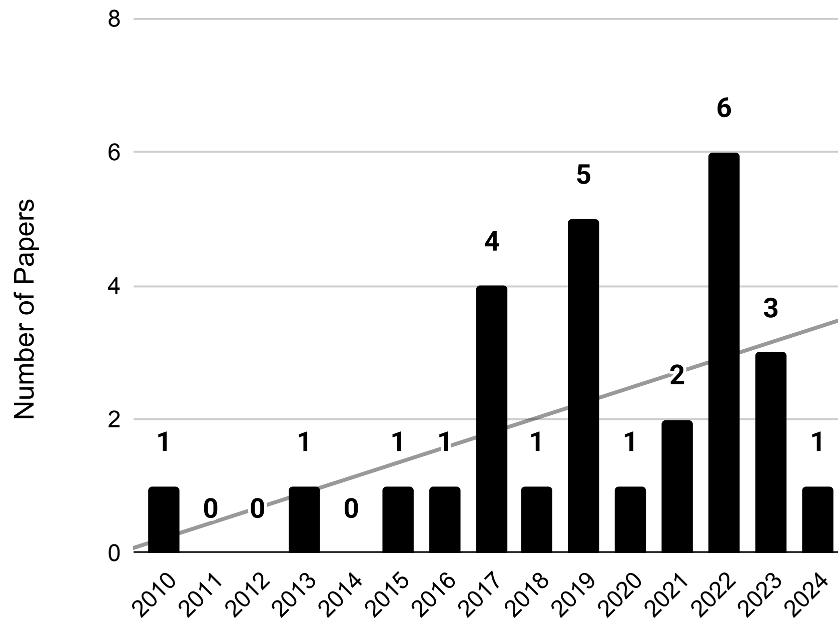



Figure 3.2: Frequency of Paper Publication by Year

Source: Elaborated by the author.

Table 3.3 lists the 27 papers in five research areas defined in this study by their main research topics: code smell detection (9 papers), JavaScript-specific code quality (8 papers), static analysis and linters (6 papers), test smell detection (2 papers), and mining software repositories (2 papers). Code smell detection has progressed from early static analysis [28] to framework-specific studies [30] and client-server smell detection [91]. JavaScript-specific quality research explores TypeScript maintainability [14], async programming issues [116], and bug benchmarking [42]. Static analysis and linters research

examines linter usage [111] and dynamic tools like DLint [37], highlighting the need for adaptive linting. Test smell detection remains under-explored but it reveals issues such as asynchronous test failures [51]. Lastly, mining software repositories use GitHub data to assess JavaScript quality trends [15], reinforcing the value of empirical validation for practical development.

Table 3.3: Research Focus Areas and Paper Classification

| **Research Focus Area** | **#** | **Papers List** |
|---|---|---|
| Code Smell Detection | 9 | JSNOSE: Detecting JavaScript Code Smells [28]<br>A Large-scale Empirical Study of Code Smells in JavaScript Projects [50]<br>An Empirical Study of Code Smells in JavaScript Projects [93]<br>On the Use of Smelly Examples to Detect Code Smells in JavaScript [102]<br>Detecting Code Smells in React-based Web Apps [30]<br>Causal Inference of Server- and Client-side Code Smells in Web Apps Evolution [91]<br>Are Existing Code Smells Relevant in Web Games? an Empirical Study [53]<br>Automated Refactoring of Legacy JavaScript Code to ES6 Modules [83]<br>Code Smell Detection Tool for JavaScript Programs [4] |
| JavaScript-Specific Code Quality Issues | 8 | To type or not to type? A Systematic Comparison of the Software Quality of Javascript and Typescript Applications on Github [14]<br>How and Why We End Up With Complex Methods: a Multi-language Study [61]<br>Anomaly Detection in Dynamic Programming Languages Through Heuristics Based Type Inference [48]<br>BUGSJS: A Benchmark of JavaScript Bugs [42]<br>Detecting Unknown Inconsistencies in Web Applications [74]<br>Detecting Common Weakness Enumeration Through Training the Core Building Blocks of Similar Languages Based on the CodeBERT Model [84]<br>DrAsync: Identifying and Visualizing Anti-patterns in Asynchronous JavaScript [116]<br>JSOptimizer: An Extensible Framework for JavaScript Program Optimization [58] |
| Static Analysis & Linters | 6 | DLint: Dynamically Checking Bad Coding Practices in JavaScript [37]<br>The Adoption of Javascript Linters in Practice: A Case Study on ESLint [111]<br>Analyzing Linter Usage and Warnings Through Mining Software Repositories: A Longitudinal Case Study of JavaScript Packages [45]<br>Let Us Lint: A Tool for Code Formatting And Code Enhancing [119]<br>JSGuide: A Tool to Improve JavaScript Algorithms Focusing on IoT Devices [75]<br>GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications [39] |
| Mining Software Repositories | 2 | Mining Rule Violations in JavaScript Code Snippets [15]<br>Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild [85] |
| Test Smell Detection | 2 | Investigating Test Smells in Javascript Test Code [51]<br>Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection [35] |

Figure 3.3 summarizes this result by showing a growing academic interest in JavaScript code smell detection. Early studies (2013–2017) focused on static analysis and heuristics, appearing in specialized venues, such as SANER and SCAM. Recent research (2019–2024) has gained visibility in major conferences, such as ICSE, FSE, and ASE, suggesting stronger empirical evaluation. Journals, such as EMSE and TSE, have published studies on JavaScript quality, including repository analyses, expanding the field

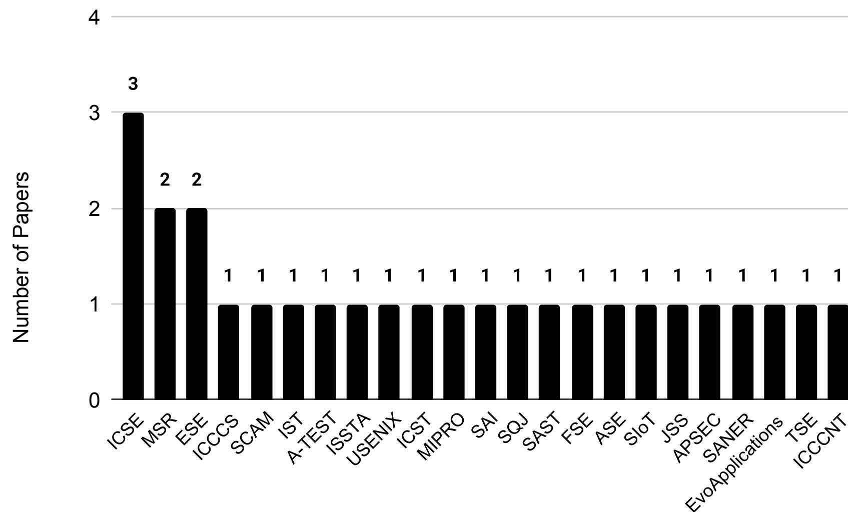



Figure 3.3: Frequency of the Topic Discussion in Conferences and Journals

Source: Elaborated by the author.

beyond niche discussions. Research on static analysis tools and empirical smell detection has influenced industry practices, particularly in linters and automated code quality tools.

Although research on JavaScript code smell detection has grown, important gaps still exist. For example, test smell detection remains underexplored, with only two papers addressing it. Similarly, machine learning-based approaches, which are common in Java and Python [9, 110, 118], are still rare in JavaScript. This is surprising, given the success of AI techniques in other programming ecosystems. One reason for this gap may be the lack of benchmark datasets for JavaScript code smells, which makes it harder to train and evaluate models. This dissertation focuses on AI-driven detection, especially using LLMs to detect code smells. We also address the mining of real-world repositories, which can help build realistic datasets and uncover practical smell patterns.

> **$\mathbf{RQ}_1$ Findings:** JavaScript code smell detection research has progressed from static analysis and heuristics to framework-specific and empirical studies. Although linters and static analysis tools have influenced industry best practices, we still lack test smell detection, AI-driven techniques, and refactoring tools focused in JavaScript. The 27 analyzed papers were published in top-tier venues (ICSE, FSE, ASE), specialized conferences (SANER, MSR, SCAM), and journals (TSE, ESE).

## 3.5 Features of Automated Detection Strategies

With the landscape of publications on automated JavaScript code smell detection strategies established, we now analyze the data collected from primary studies. Figure 3.4 presents a list of automated strategies found in these studies along with their citation frequency. Three automated approaches stood out as the most cited: ESLint [50], JSLint [82], and JSHint [4]. The widespread adoption of JSLint and ESLint in industry, due to their versatility in code smell detection and unit testing, likely contributed to their prominence in research [111]. JSNose also gained attention, possibly because Fard and Mesbah [28] were among the earliest proposals for JavaScript code smell detection. Another notable point is the presence of general-purpose automated approaches that detect smells in multiple programming languages, such as PMD and SonarQube [102], which are widely used for Java. This result highlights their market relevance and adaptability to different languages.

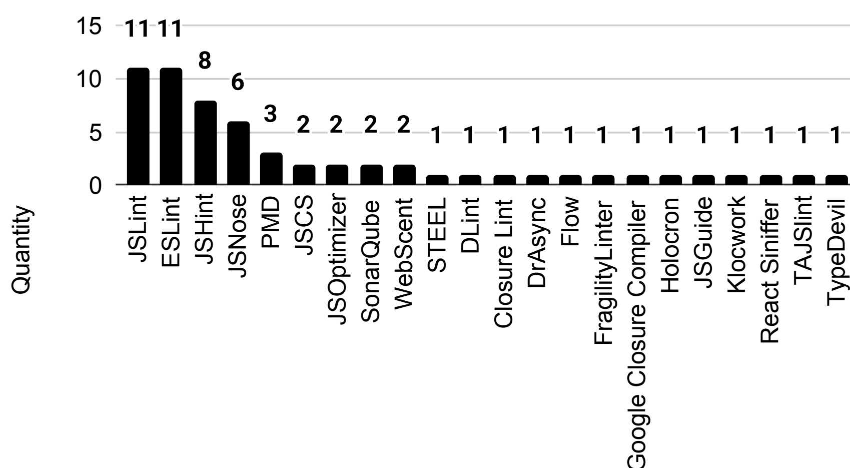



Figure 3.4: List of the Automated Strategies with Their Number of Primary Studies

Source: Elaborated by the author.

Table 3.4 lists the 22 automated code smell detection strategies identified in this SLR. Our study includes automated strategies focused on code smell detection, even if they provide additional features such as refactoring support. The findings show that widely adopted automated strategies, such as ESLint and JSLint, benefit from strong community support and seamless integration into development workflows. In contrast, less-cited automated strategies often target specific code smells or lack active community engagement. All automated strategies rely on Abstract Syntax Trees (AST) for detection, so this information is not included in the Detection Technique column.

The results show the dominance of rule-based linting, used by 55% of the auto-

Table 3.4: Automated Code Smell Detection Strategies: Smells and Detection Technique

| Strategy Name | Detected Code Smells | Detection Technique |
|---|---|---|
| Closure Linter | Coding style and formatting rules | Rule-based Linting |
| DLint | Primitive Obsession, Dead Code | Dynamic Analysis |
| DrAsync | Dead Code | Dynamic Analysis |
| ESLint | Long Method, Large Class, Data Clumps, Switch Statements and others | Rule-based Linting |
| Flow | Primitive Obsession, Dead Code | Rule-based Linting |
| FragilityLinter | Primitive Obsession | Rule-based Linting |
| Google Closure Compiler | Dead Code | Static Analysis |
| Holocron | Duplicated Code | Dynamic Analysis |
| JSCS | Coding style and formatting rules | Rule-based Linting |
| JSGuide | Rest Parameter Misuse, Spread Operator Misuse, and others | Static Analysis |
| JSHint | Long Parameter List, Switch Statements, Dead Code | Rule-based Linting |
| JSLint | Long Method, Large Class, Data Clumps, Switch Statements and others | Rule-based Linting |
| JSNose | Refused Bequest, Long Parameter List, Long Method, and others | Pattern Matching |
| JSOptimizer | String Concatenation, Loop DOM Access, and others | Call Graphs |
| Klocwork | Duplicated Code, Long Method, Large Class, Long Parameter List and others | Rule-based Linting |
| PMD | Duplicated Code, Long Method, Switch Statements and others | Rule-based Linting |
| React Sniffer | Large Component, Too Many Props, Large File, and others | Rule-based Linting |
| SonarQube | Long Method, Large Class, Lazy Class and others | Rule-based Linting |
| STEEL | Lazy Test, Duplicate Assert, Eager Test, and others | Static Analysis |
| TAJS | Dead Code | Abstract Interpretation |
| TypeDevil | Primitive Obsession | Dynamic Analysis |
| WebScent | Duplicated Code | Dynamic Analysis |

mated strategies. Popular automated approaches, such as PMD, SonarQube, JSLint, and ESLint, rely on predefined rules to analyze JavaScript code. Rule-based linting is efficient, making it the preferred method for static analysis. However, it has limitations, as it may miss deeper, more structural issues that require advanced pattern matching or runtime analysis. Although rule-based linting is the most common approach, some automated strategies go beyond it by using static analysis to improve JavaScript maintainability checks. Google Closure Compiler, STEEL, and JSGuide analyze code without running it, helping to detect issues, such as dead code, redundant operations, and improper JavaScript constructs.

However, dynamic analysis remains underused, with only 5 out of 22 automated strategies (23%) employing it. WebScent, TypeDevil, DLint, Holocron, and DrAsync execute JavaScript in controlled environments to detect runtime issues that static analysis might overlook. This approach is particularly useful for identifying dead code, type inconsistencies, and execution-based smells. Despite growing interest in AI-driven techniques

for code smell detection for other languages [107, 124], we found no automated JavaScript code smell detection strategy using Machine Learning or Large Language Models. This gap suggests an opportunity to advance JavaScript analysis with modern AI-based approaches.

Table 3.5 compares automated JavaScript code smell detection strategies based on their ability to detect specific code smells. While many automated approaches are available, most strategies do not detect higher-level object-oriented and architectural smells, as described by Fowler [34]. Instead, they primarily identify simple issues, such as Long Method, Large Class, Duplicated Code, and Dead Code. PMD, SonarQube, and JSLint focus on these basic problems, while more complex design smells, such as Shotgun Surgery and Feature Envy, are rarely detected. Among the automated strategies analyzed, JSNose is one of the few capable of detecting Refused Bequest, a design smell related to improper inheritance in object-oriented programming. This lack of deep architectural smell detection suggests that most automated JavaScript analysis strategies prioritize syntax and performance issues over maintainability and design complexity.

Table 3.5: Detected Code Smells by Automated JavaScript Detection Strategies

| **Code Smells** | **Automated Strategies** |
|---|---|
| Dead Code | Google Closure Compiler, TAJS, JSHint, ESLint, Flow, DLint, DrAsync |
| Duplicated Code | Klocwork, PMD, WebScent, Holocron |
| Large Class | Klocwork, SonarQube, JSLint, ESLint |
| Long Method | Klocwork, PMD, SonarQube, JSLint, ESLint, JSNose |
| Long Parameter List | Klocwork, JSHint, JSNose |
| Primitive Obsession | TypeDevil, DLint, Flow, FragilityLinter |
| Refused Bequest | JSNose |
| Switch Statements | PMD, JSHint, JSLint, ESLint |

Table 3.6 shows some important features of the 22 automated code smell detection strategies identified in this SLR. A notable observation is that most automated approaches are command-line-based, with only 4 out of 22 automated strategies (18%) providing a Graphical User Interface (GUI). These automated strategies include: Klocwork, SonarQube, JSLint, and FragilityLinter. The lack of GUI support in most automated strategies indicates that they are primarily designed for integration into development pipelines rather than standalone interactive usage. Developers must rely on IDE extensions, linters, or command-line outputs to interpret their results.

With the exception of Klocwork, all automated strategies we found are free for use (95% are open-source). This suggests that the JavaScript development community heavily relies on open-source solutions to improve code quality. The availability of automated open-source approaches also facilitates continuous improvements as new JavaScript fea-

Table 3.6: Automated Detection Strategies: Language, Availability and Features

| Strategy Name | Language | Plug-in | Free | Guide | GUI | Year | Maint. | IoR |
|---|---|---|---|---|---|---|---|---|
| FragilityLinter | JavaScript | Yes | Yes | No | Yes | 2022 | No | Research |
| JSGuide | Java | No | Yes | No | No | 2022 | No | Research |
| STEEL | JavaScript | No | Yes | No | No | 2021 | No | Research |
| DrAsync | JavaScript | No | Yes | Yes | No | 2021 | No | Research |
| React Sniffer | JavaScript | No | Yes | Yes | No | 2021 | No | Research |
| JSOptimizer | JavaScript | No | Yes | No | No | 2019 | No | Research |
| Holocron | JavaScript | No | Yes | No | No | 2017 | No | Research |
| TypeDevil | JavaScript | No | Yes | No | No | 2015 | No | Research |
| Flow | OCaml | Yes | Yes | Yes | No | 2015 | Yes | Industry |
| Closure Linter | Python | No | Yes | Yes | No | 2015 | No | Industry |
| DLint | JavaScript | No | Yes | Yes | No | 2015 | No | Research |
| JSLint | JavaScript | Yes | Yes | Yes | Yes | 2013 | Yes | Industry |
| ESLint | JavaScript | Yes | Yes | Yes | No | 2013 | Yes | Industry |
| JSCS | JavaScript | Yes | Yes | Yes | No | 2013 | No | Industry |
| JSNose | Java | No | Yes | Yes | No | 2013 | No | Research |
| WebScent | PHP | No | Yes | No | No | 2012 | No | Research |
| JSHint | JavaScript | Yes | Yes | Yes | No | 2011 | Yes | Industry |
| Google Closure Compiler | Java | No | Yes | Yes | No | 2009 | No | Industry |
| TAJS | Java | No | Yes | Yes | No | 2009 | No | Research |
| SonarQube | Java | Yes | Yes | Yes | Yes | 2007 | Yes | Industry |
| PMD | Java | Yes | Yes | Yes | No | 2002 | Yes | Industry |
| Klocwork | C++ | Yes | No | Yes | Yes | 2001 | Yes | Industry |

tures and frameworks emerge. However, of the 22 analyzed automated strategies, only 7 still receive updates. This suggests a general lack of long-term support for many automated approaches in the ecosystem. However, an interesting pattern emerges when looking at the two most cited automated strategies, ESLint and JSLint, both of which are still maintained. Their active maintenance may partly explain their popularity, as developers tend to adopt automated approaches that remain up-to-date with evolving language features and development practices.

An important observation is that among the 22 identified automated approaches, exactly half were developed in an academic or research context, while the other half were created by the software industry. This balance highlights how both communities have contributed to JavaScript code smell detection, but the outcomes and impact of these efforts differ in important ways. One major difference is that all automated strategies that are still maintained come from the industry. In contrast, none of the research-developed automated approaches continue to receive updates. This suggests that industry automated strategies are more likely to be sustained over time, likely due to broader adoption, funding, and integration into real development environments. ESLint, JSLint, and SonarQube are still actively used and maintained. They have strong community or company support and are designed for daily use in production systems.

Lastly, 10 of the 22 automated strategies (45%) function as plug-ins, including

JSHint, ESLint, JSLint, and PMD. These automated approaches are integrated into IDEs, such as VSCode and JetBrains, providing real-time linting and code quality checks. The remaining 12 automated strategies (55%) operate as standalone CLI tools, primarily used in CI/CD pipelines rather than for direct developer feedback. This distinction highlights a trade-off: plug-ins offer instant feedback for developers, while automated standalone approaches enable deeper, batch-style analysis in automated workflows.

Table 3.7 shows the trends that can be observed from the release year of the automated strategies. A trend towards newer automated approaches focuses on modern JavaScript frameworks and features. Older automated strategies such as JSHint, PMD, and SonarQube mainly detect general JavaScript code smells. In contrast, newer automated approaches, such as React Sniffer, JSGuide, and FragilityLinter, analyze code based on modern development practices. For example, React Sniffer detects React-specific smells, such as Large Component, Too Many Props, and Large File, emphasizing the need for framework-aware static analysis. Similarly, JSGuide identifies misuse of ECMAScript 2015 features, such as Rest Parameter Misuse and Spread Operator Misuse, highlighting a shift towards analyzing newer JavaScript features.

Table 3.7: Evolution of Code Smell Detection over Time

| **Period** | **Trend** |
|---|---|
| 2001-2010 | Basic rule-based analysis (Klocwork, PMD, SonarQube) |
| 2010-2015 | Expansion into dynamic analysis, type checking, and more specific smells (JSHint, WebScent, JSNose, TypeDevil, Flow) |
| 2016-2022 | Focus on framework-specific smells, ECMAScript 2015 or newer features, and test smells (STEEL, DrAsync, React Sniffer, FragilityLinter, JSGuide) |

**RQ$_2$ Findings:** Our review of 22 automated JavaScript code smell detection strategies shows that most use rule-based linting to find basic issues such as long methods, large classes, and duplicated code. Only JSNose detects more complex smells, such as refused bequest, and few automated approaches address design problems. Test smell detection is growing, with automated strategies such as STEEL and FragilityLinter focusing on test code. Newer automated strategies, such as React Sniffer and JSGuide, target modern JavaScript frameworks. There are still gaps in AI use and code smells detection, which we address in the next chapters.

## 3.6 Threats to Validity

We relied on strict guidelines [127] to discuss threats to the study validity as follows.

**Construct Validity:** We used an extensive search string that included some of the most common terms related to code smells in JavaScript. One potential threat to this approach is the possibility that the search string might not cover a sufficient number of primary studies. To mitigate this possible threat, we created the search string iteratively and in pairs and through multiple tests on real Web search engines. Another potential threat lies in the inclusion and exclusion criteria chosen for primary studies. To avoid deviating from the context of our study, we decided to exclude incomplete automated strategies or those that were heuristic in nature, which may have led to the exclusion of some strategies. Finally, a potential threat lies in the fact that only the metadata was used to refine papers coming from the electronic data sources. To mitigate this possible threat, we performed both backward and forward snowballing, allowing us to collect papers that might have been lost on refinement.

**Internal Validity:** This study considers possible threats related to the data collection process. One threat involves the risk of incorrect data export and tabulation from the Web search engines. To mitigate this threat, we adopted a paired data collection approach in which one researcher monitored and verified the work of another. This collaborative process helped reduce the likelihood of errors and ensured greater reliability in data extraction and organization.

**External Validity:** We decided to focus our work on automated strategies specifically designed for software systems built using JavaScript rather than the so-called language-agnostic approaches. This decision may have significantly limited our results, reducing the number of automated approaches identified. To mitigate this threat, we used the guidelines of a SLR [55], which describe that industrial automated strategies not published in papers may be removed from the list of study objects.

## 3.7 Chapter Summary

In this chapter, we detailed our SLR on automated strategies for code smell detection. We focus on JavaScript because we found evidence of low adaption of AI techniques for this language. Academic interest in this area has grown, and more studies have appeared in top venues, such as ICSE, MSR, and TSE. We identified 22 automated

strategies, with ESLint, JSLint, and JSHint being the most widely cited and adopted. Rule-based linting dominates (55% of strategies), but often misses complex architectural smells, while dynamic analysis is less common, and no automated strategy has yet used AI-driven techniques. Most automated strategies are open-source (95%), and the ecosystem is split between CLI-based strategies (82%) and IDE plug-ins (45%), showing a balance between real-time feedback and automated workflow integration.

In the next chapters, we focus on the use of LLMs for automating code smell detection. Chapter 4 describes the original dataset used in our empirical evaluation of code smell detection with LLMs and explain how we expanded it. Finally, we outline the process of creating the automated ground truth, including the selection of LLMs, the sampling of code smells, the prompts used, and the voting procedure.

# Chapter 4

# A Dataset of Automatically Detected Code Smells

Several code smells datasets are available in the literature covering a wide range of programming languages [20, 63, 109]. However, they tend to have several limitations. For instance, systems in some datasets may not represent current development practices [20, 109] or they present limited coverage of code smells [63]. To avoid these limitations, we have extended a dataset from the literature [94]. Using the created dataset, we built an automated ground truth using a majority voting strategy that combines the output of static analysis tools and LLMs. This automated ground truth is important as it provides a consistent and scalable benchmark for evaluating the effectiveness of different code smell detection approaches. This chapter presents the dataset created for this dissertation, which integrates the outputs of four LLMs for code smell detection.

The chapter is structured as follows. Section 4.1 describes the original dataset used as the starting point of our study with LLMs. Section 4.2 outlines the first two steps of the automated ground truth creation, including the selected LLMs and sampling of the code smells. Section 4.3 describes the prompt we used. Section 4.4 outlines the voting strategy that led to our automated ground truth. Section 4.5 discusses some threats to the study validity. Finally, Section 4.6 concludes this chapter and introduces the next one.

## 4.1 The Original Dataset

We extended a dataset of code smells from the literature [94]. Table 4.1 provides details on the 30 systems included in the original dataset [94]. The first column lists the names and versions of each system. The next four columns show the number of classes (NOC), number of methods (NOM), the lines of code (LOC), and number of code stars (NS) for each repository in the original dataset [94], respectively. Data collection for the

original dataset spanned from April 2021 to April 2023. The selected dataset included 3,459 instances of nine code smells (three at the class level and six at the method level) identified in 30 open-source Java systems from GitHub. We focus on projects in Java for our study with LLMs because many tools are available to detect a wide range of code smells in this programming language [29] along with a better availability of datasets in the literature [20, 63, 109].

Table 4.1: Systems in the Original Dataset

| **Name** | **NOC** | **NOM** | **LOC** | **Stars** |
|---|---|---|---|---|
| arthas-3.4.3 | 834 | 4,733 | 39,973 | 36,580 |
| cryptomator-1.6.1 | 590 | 2,690 | 16,350 | 13,388 |
| dbeaver-21.0.2 | 6,449 | 36,575 | 348,608 | 44,710 |
| easyexcel-2.2.11 | 249 | 1,629 | 10,639 | 33,549 |
| elasticsearch-analysis-ik | 28 | 203 | 2,051 | 17,200 |
| fastjson-1.2.76 | 249 | 1,996 | 44,434 | 25,756 |
| gson-2.8.8 | 231 | 924 | 11,721 | 23,925 |
| guava-30.1.1 | 27,412 | 200,616 | 2,125,859 | 50,993 |
| HikariCP-4.0.0 | 68 | 581 | 4,530 | 20,634 |
| hutool-5.7.17 | 1,214 | 11,966 | 79,432 | 29,969 |
| java-faker-1.0.2 | 105 | 751 | 3,602 | 4,889 |
| jedis | 749 | 6,219 | 27,404 | 12,132 |
| jenkins-2.287 | 2,432 | 14,775 | 120,382 | 24,269 |
| jitwatch-1.4.2 | 538 | 7,346 | 46,527 | 3,185 |
| jsoup-1.14.2 | 246 | 1,551 | 17,449 | 11,235 |
| junit4-4.13.2 | 310 | 1,541 | 10,769 | 8,533 |
| libgdx-gdx-1.9.14 | 2,714 | 39,338 | 208,028 | 1,263 |
| mall-1.0.2 | 747 | 14,322 | 100,990 | 81,223 |
| mybatis-3.5.6 | 378 | 2,582 | 20,533 | 20,179 |
| nanohttpd-2.3.1 | 75 | 405 | 3,821 | 7,129 |
| netty-socketio-1.7.18 | 138 | 712 | 5,217 | 6,980 |
| redisson-3.15.3 | 1,613 | 13,607 | 82,104 | 23,946 |
| retrofit-1.6.0 | 118 | 403 | 4,790 | 43,653 |
| rocketmq-4.9.2 | 996 | 7,536 | 70,871 | 21,990 |
| Sa-Token-1.28.0 | 191 | 1,600 | 8,848 | 18,043 |
| Sentinel-1.8.3 | 1,029 | 4,963 | 41,366 | 22,852 |
| spring-cloud-alibaba-2.2.2 | 411 | 2,003 | 13,594 | 28,662 |
| webmagic-develop-0.7.6 | 207 | 955 | 6,757 | 11,600 |
| xxl-job-2.3.0 | 150 | 741 | 8,374 | 29,124 |
| zxing-3.4.1 | 303 | 1,783 | 23,614 | 33,495 |
| **Total** | 50,774 | 385,046 | 3,508,637 | 682,312 |

The original dataset [94] relies on the following criteria to select the systems to compose: (i) they were among the top-star Java systems on GitHub; (ii) they had commits merged in the last two years; and (iii) they cover different domains, sizes, and levels of

maturity. Systems for educational purposes were excluded [94]. This variety of systems helps us investigate how well LLMs can detect code smells in software systems of different domains and sizes. Individual outputs of four static analysis tools – JDeodorant, JSpIRIT, Organic, and PMD – are also available in the original dataset [94] and we used them to support the comparison with LLMs. These tools were chosen because they have shown good accuracy in detecting code smells in previous studies [20, 29, 31, 63, 77, 95].

Table 4.2 lists the static analysis tools used in the previous work [94] to detect each of the 9 smells considered in their study: Data Class (DC), Dispersed Coupling (DiCo), Feature Envy (FE), Intensive Coupling (IC), Long Method (LM), Large Class (LC), Long Parameter List (LPL), Refused Bequest (RB), and Shotgun Surgery (SS). A "✔" mark in the table means that the tool was used to detect the smell shown in the respective column. In the original dataset [94], the researchers applied a voting strategy to mitigate tool bias, using two tools to detect each smell. For example, JDeodorant and JSpIRIT were used to detect Large Class (LC).

Table 4.2: Static Analysis Tools Used to Detect Code Smells

| **Tool** | **DC** | **DiCo** | **FE** | **IC** | **LC** | **LM** | **LPL** | **RB** | **SS** |
|---|---|---|---|---|---|---|---|---|---|
| JDeodorant | | | ✔ | | ✔ | ✔ | | | |
| JSpIRIT | | ✔ | | ✔ | ✔ | | | ✔ | ✔ |
| Organic | ✔ | ✔ | ✔ | ✔ | | ✔ | ✔ | ✔ | ✔ |
| PMD | ✔ | | | | | | ✔ | | |

## 4.2 Automated Ground Truth Creation

Figure 4.1 presents the steps we followed to automatically create a ground truth, which is used in this dissertation to evaluate the effectiveness of LLMs in detecting code smells. Together, these steps ensure a systematic and reproducible process for constructing the automated ground truth. The four steps are numbered from 1 to 4 to make it easier we refer to them in the text, ensuring that each step can be understood in the context of the overall study. This section describes the first two steps, namely Select LLMs and Sample Smell Candidates. We discuss the other two steps, the Used Prompt and the Voting Strategy of Tools and LLMs, in Sections 4.3 and 4.4, respectively.

**Selected Large Language Models:** The first step of our study (Step 1 in Figure 4.1) is to select the LLMs. Table 4.3 shows important information about the LLMs selected for this study: OpenAI's GPT-4o, Meta's Llama-3.3-70B-Instruct, DeepSeek's

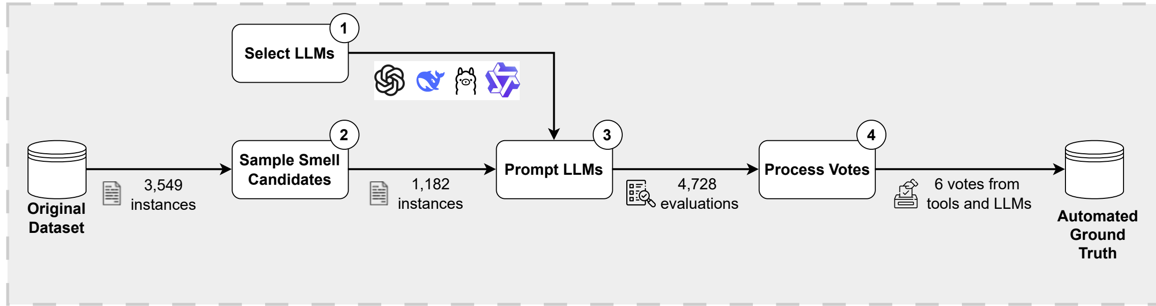



Figure 4.1: Steps of the Automated Ground Truth Creation

Source: Elaborated by the author.

DeepSeek-R1-Distill-Qwen-32B, and Qwen's Qwen2.5-Coder-32B-Instruct. All models have similar capabilities. For instance, DeepSeek-R1 and Qwen2.5-Coder both feature 32.5 billion parameters and a large context window of 131,000 tokens. Their knowledge cutoffs are July 2024 and March 2024, respectively. Llama-3.3 has 70 billion parameters and a context window of 128,000 tokens. Although GPT-4o does not disclose its parameter count, it has a similar context window of 128,000 tokens and a slightly earlier knowledge cutoff date of September 2023.

Table 4.3: Selected Large Language Models

| Model | Parameters | Context Window | Knowledge Cutoff |
|---|---|---|---|
| DeepSeek-R1 | 32.5B | 131K | July 2024 |
| GPT-4o | N/A | 128K | Sep 2023 |
| Llama-3.3 | 70B | 128K | Dec 2023 |
| Qwen2.5-Coder | 32.5B | 131K | Mar 2024 |

We chose these LLMs because they are often used in recent research [44, 46, 64, 134] and are popular among developers. For instance, GPT models have attracted a lot of attention from researchers, with studies covering many topics of software engineering, such as code generation [59], computer science education [129], refactoring [24], code review [40], test case generation [132], and library selection [108]. We also included Llama-3.3 and DeepSeek-R1 because they are open source, which allows us to compare them with closed-source models like GPT-4o. In July 2025, both open-source models had been downloaded more than 400,000 times on HuggingFace, showing the strong interest from the developer community. In addition, we include an open-source model focused on code generation, Qwen2.5-Coder, to bring a different point of view to our study. Qwen2.5-Coder was the highest-ranked code model on Hugging Face's leaderboard in July of 2025 [1].

[1] https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard

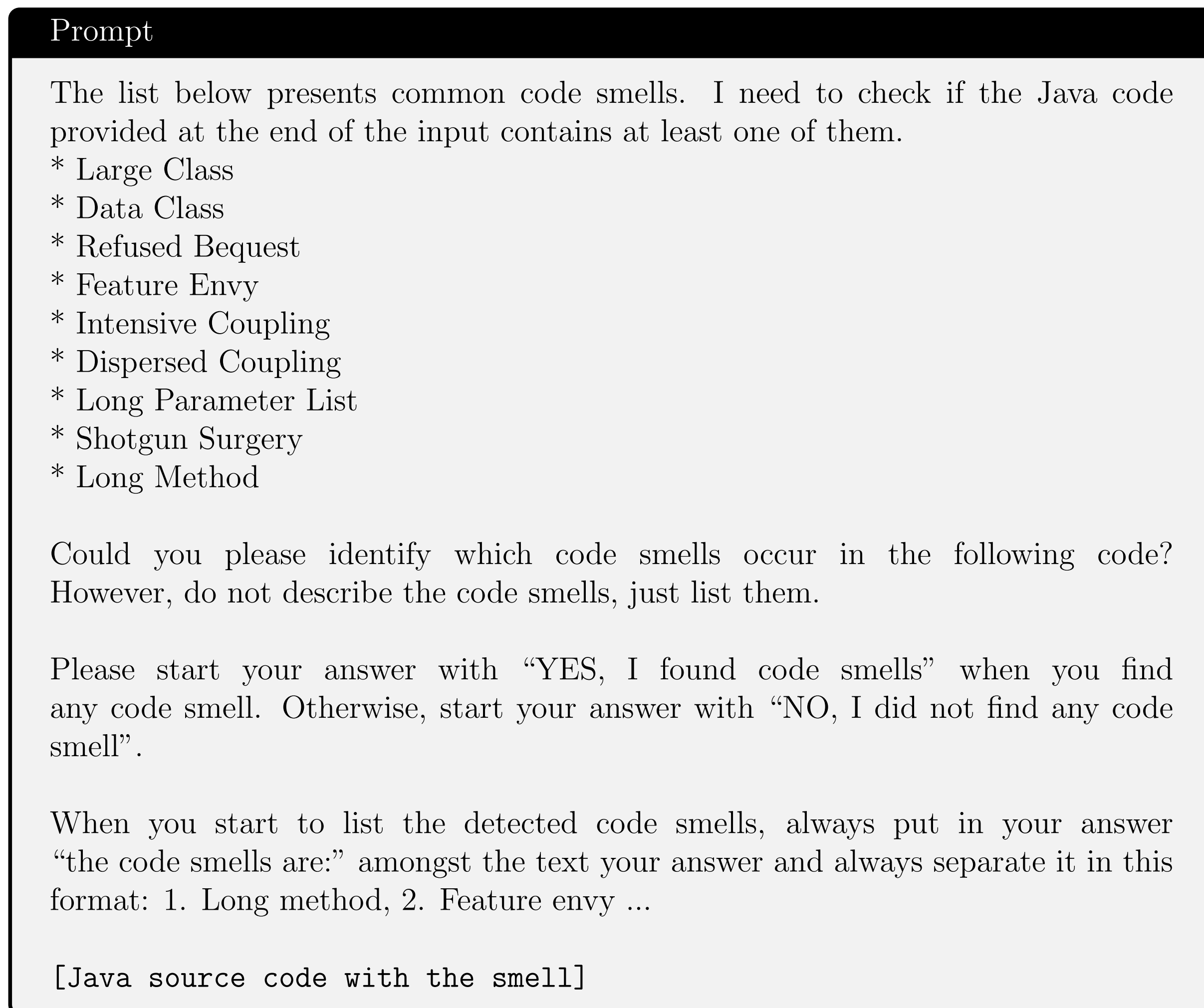
**Sampling Code Smells:** The second step of our study (Step 2 in Figure 4.1) was to sample code smells. We randomly selected 1,182 instances of code smells as candidates for the automated creation of the ground truth, which represents about 34% of all instances in the original dataset [94]. We also sampled 268 instances for human evaluation in our empirical study, further detailed in Chapter 5. This sample is a subset of the previous one, allowing us to assess the reliability of the automated ground truth creation. A smaller sample was selected for human evaluation due to constraints on enrolling a large group of participants to evaluate the code smell instances manually.

The two samples included both smelly and non-smelly classes and methods [94]. They are both balanced and allow both static analysis tools and LLMs to distinguish smelly from non-smelly instances. Although six code smells in this study are found at the method level, we decided to provide the entire class, including the smelly methods, to the LLMs. This gives the models more context to help them detect code smells. Such context information could be particularly important for some smells relying on the inheritance relationship (e.g., Feature Envy) and class dependencies (e.g., Shotgun Surgery).

## 4.3 Used Prompt

This section explains the third step (Prompt LLMs) for the automated ground truth creation. Figure 4.2 shows the structured prompt we use in this step. Inspired by a previous work [103], this prompt clearly lists the nine code smells we aim to detect. It relies on zero-shot learning [123]; i.e., it does not provide the LLM with any example of code smells. Instead, the prompt simply asks the model to perform the task without further details. In addition to being widely used in research [46], zero-shot prompt simulates how developers actually interact with an LLM in real settings. It is also important to note that we do not aim to evaluate different prompts and fine-tuning strategies. Similar to static analysis tools, a zero-shot prompt is simpler for developers and relies on the basic capabilities of the models. Since we clearly specified the output format in the prompt, we were able to process them automatically. That is, when a response included "YES", we automatically extracted the code smells by reading the text after the phrase "The code smells are:". If the model did not find any code smells, the standard output was "NO, I did not find any code smell".

To quantify the number of code smells in each class, we recorded "1" for every code smell listed in the output and "0" otherwise. For classes where the models were unable to detect any smell, we recorded "0" for all smells. Finally, to make all outputs consistent and comparable, the temperature was set at 0.0 for all models, except DeepSeek-R1. For



Prompt

The list below presents common code smells. I need to check if the Java code provided at the end of the input contains at least one of them.
* Large Class
* Data Class
* Refused Bequest
* Feature Envy
* Intensive Coupling
* Dispersed Coupling
* Long Parameter List
* Shotgun Surgery
* Long Method

Could you please identify which code smells occur in the following code? However, do not describe the code smells, just list them.

Please start your answer with "YES, I found code smells" when you find any code smell. Otherwise, start your answer with "NO, I did not find any code smell".

When you start to list the detected code smells, always put in your answer "the code smells are:" amongst the text your answer and always separate it in this format: 1. Long method, 2. Feature envy ...

`[Java source code with the smell]`

Figure 4.2: Prompt Used for Detecting Code Smells

Source: Elaborated by the author based on the work of Silva et al. [103].

this model, after failing to get useful responses at 0.0 temperature, we changed it to 0.6 as recommended by its HuggingFace instructions [2]. We also limited the output of all LLMs to 300 characters. Of 4,728 prompts across all models, 450 (9%) either exceeded the maximum token limit or could not produce an expected output within the 300-character limit. DeepSeek-R1 presented most issues; it failed in 438, GPT-4o failed in 9, and Qwen2.5 failed in 2 out of 1,182 prompts for each LLM. Only Llama-3.3 successfully provided the expected output for all prompts.

[2] https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B

## 4.4 Voting Strategy of Tools and LLMs

This section explains how we automatically create a ground truth for our study. In Step 4 of Figure 4.1, we used a voting strategy [94] that combines the results for each code smell from two static analysis tools and four LLMs (6 votes). We should note that, although we rely on outputs of four tools, only two of them were used for each smell as detailed in Section 4.1. Following a majority voting strategy, if a class receives four votes for a code smell, then we add it to our ground truth for that code smell. All votes have the same weight and, therefore, tools and LLMs are equally able to influence the final decision of a smell candidate.

Table 4.4 presents the distribution of code smells in the ground truth according to the 1,182 evaluated classes. Based on this voting strategy, the most common code smell is Long Method, which appears in 616 classes (52%) of the dataset. Large Class is the second most frequent, with 255 instances (21%). The other code smells, such as Long Parameter List, Intensive Coupling, Data Class, Feature Envy, and Dispersed Coupling, are much less common, each representing fewer than 10% of the dataset classes. Interestingly, we could not find any instance of Refused Bequest and Shotgun Surgery by using the voting strategy. This result may indicate either that these code smells are rare in the sampled projects or suggest possible limitations in the tools and LLMs used to detect them. In line with previous work [16, 31, 62, 77], our ground truth shows an imbalance in the distribution of code smells.

Table 4.4: Code smells in the Automated Ground Truth

| Code Smell | Number of Instances | Percentage of Instances |
| --- | --- | --- |
| Data Class | 49 | 4% |
| Dispersed Coupling | 23 | 2% |
| Feature Envy | 43 | 4% |
| Intensive Coupling | 61 | 5% |
| Large Class | 255 | 21% |
| Long Method | 616 | 52% |
| Long Parameter List | 72 | 6% |
| Refused Bequest | 0 | 0% |
| Shotgun Surgery | 0 | 0% |

## 4.5 Threats to Validity

This section discusses the main validity concerns related to the creation of our dataset and its automated ground truth, including internal, external, construct validity, and reliability. We describe potential threats in each area and outline the steps taken to mitigate them.

**Internal validity** concerns factors that could affect the results of our study without our knowledge [127]. An important threat is the sensitivity of the prompts used for the LLMs and their configuration since small changes in prompts or settings can lead to different results, making it difficult to attribute outcomes solely to the model's capabilities. To mitigate this, we used the same prompt design for all LLMs, the temperature recommended by the models creators to achieve the most deterministic answers, reported all configuration details, and provided all prompts in the replication package [3] to ensure findings are robust and reproducible. Another threat comes from the eventual bias introduced by the code samples used by LLMs during their training phase, potentially impacting their ability to detect code smells in previously unseen code. We mitigated this threat by using a dataset that contains source code extracted from the 30 most starred GitHub repositories. By selecting widely known code samples, we ensure that all models are equally likely to have been exposed to the same code samples during their training step. This approach promotes consistency in the evaluation.

**External validity** addresses the extent to which our findings can be generalized beyond the specific context of our study [126]. Generalization to other languages and contexts is a threat to external validity, as results based solely on Java projects may not apply to other programming languages or software domains. Another important aspect concerns the generalization to other LLMs. In this study, we selected four representative models, but the landscape of LLMs evolves rapidly, and different architectures, training data, or fine-tuning strategies may lead to different outcomes. Similarly, generalization to other static analysis tools is limited, since we relied on a subset of widely used tools, and other tools may implement different detection heuristics or rules that could affect agreement and effectiveness. Finally, generalization to other code smells is also a concern, as our dataset focused on nine well-known smells, but many additional smells exist in the literature, which may present different detection challenges. Although our study faces certain generalization constraints, we present a rigorous and well-documented design that enables future replications to incorporate additional detection tools, software systems, programming languages, and evaluation metrics.

**Construct validity** threat comes from how the ground truth is established, such

[3] https://zenodo.org/records/16790193

as using a voting strategy and the selection of agreement metrics, both of which can introduce bias into the observations. To mitigate this, we grounded our study design in established practices and recommendations adopted by other peer-reviewed studies, ensuring that it is aligned with widely accepted approaches in the field [3, 38, 86, 90]. The decision to treat all LLMs' missing or non-responding answer as negative ones (0) also poses a threat. Basically, this approach assumes that the lack of response is equivalent to the absence of a code smell, which may not be accurate since non-responses are often due to technical limitations, i.e., the model not being able to consistently answer within the output limit. We document this decision in our study design and ensure that the same rule was consistently applied in all scenarios.

## 4.6 Chapter Summary

In this chapter, we presented the dataset and study design for evaluating LLMs in code smell detection. To address the shortcomings of previous datasets, we expanded an existing collection of 3,459 instances of nine code smells from 30 diverse and widely used Java projects, incorporating LLM-based detection for a subset of 1,182 smells. By combining the outputs of four static analysis tools and four state-of-the-art LLMs through a majority voting strategy, we established an automated ground truth, using a zero-shot prompt to reflect developer interactions. Our results reveal that some smells, such as Long Method and Large Class are more common, while more complex smells are much less frequent.

In the next chapter, we focus on empirically evaluating how well different LLMs detect code smells, comparing their performance to static analysis tools. We outline the study protocol, including our objectives and research questions, and present a thorough discussion of the results.

# Chapter 5

# Evaluating the Effectiveness of LLMs for Code Smell Detection

Recent advances in LLMs have sparked interest in their use for coding tasks [2, 67, 73], including code generation [52], bug repair [19], and software testing [122]. However, their application to code smell detection is still underexplored and lacks strong benchmarks [128]. Unlike traditional static analysis tools, such as JDeodorant [113] and PMD [87], LLMs can offer more flexible, context-aware strategies. Although some early studies have examined LLMs for code smell detection [103], they mostly focus on small, controlled examples [21, 59, 115], and there is little empirical evidence if LLMs can outperform static analysis tools in real-world projects [128]. As real software projects introduce challenges, such as large codebases and diverse standards [72], it is essential to further investigate how LLMs compare to static analysis tools and how they address both technical and human perspectives on code smells. In this chapter, we address this issue by evaluating the effectiveness of LLMs by using four models, namely GPT-4o, Llama-3.3, DeepSeek-R1, and Qwen2.5-Coder, to detect nine code smells in 30 real-world Java projects. Our goal is to understand which code smells LLMs are better able to detect and when they fail.

The chapter is structured as follows. Section 5.1 introduces the study goal and research questions that guide our empirical investigation. Section 5.2 details the evaluation steps, including the construction of ground truths and the design of the empirical study. Section 5.3 reports the results of the agreement analysis, showing how LLMs compare with each other and with static analysis tools in different code smells. Section 5.4 evaluates the effectiveness of LLMs against an automated ground truth, highlighting their strengths and weaknesses in code smell detection. Section 5.5 examines the effectiveness of LLMs in a human ground truth, providing insight on how well they align with the perception of code smells by developers. Section 5.6 discusses some threats to the study validity and the measures we adopted to mitigate them. Finally, Section 5.7 summarizes the main findings of the chapter and introduces the next one.

## 5.1 Goal and Research Questions

This study aims to evaluate the effectiveness of LLMs in detecting code smells in software projects, comparing their performance with both traditional static analysis tools and human judgment. Our primary goal is to understand not only whether LLMs can detect code smells, but also how their performance compares to existing approaches in terms of agreement (Cohen's Kappa), precision, recall, and F1-score. To achieve this goal, we defined the following three research questions ($\text{RQ}_S$).

- **$\text{RQ}_1$:** To what extent does an LLM agree with other models and with static analysis tools in code smell detection?
- **$\text{RQ}_2$:** How effective are LLMs in detecting code smells with respect to an automatically generated ground truth?
- **$\text{RQ}_3$:** How effective are LLMs compared to human-detected code smells?

## 5.2 Evaluation Steps

Figure 5.1 presents the steps taken to evaluate LLMs for code smell detection, aiming at answering our research questions. After we automatically created a ground truth of code smells based on the majority votes of tools and LLMs, as described in Chapter 4, we also performed three steps (Steps 1 to 3) to create a human-centered ground truth. The reason is to evaluate whether LLMs comply not only with other automated strategies but also with human perspectives of the analyzed code smells. As shown in Step 1 of Figure 5.1, we first selected 76 participants to manually evaluate the code smell candidates. These participants are undergraduate students in their last year of Computer Science. A background assessment confirmed that they have sufficient experience with Java programming. Furthermore, we provided them with the necessary knowledge about code smells before their participation.

We applied stratified random sampling [8] to select 268 code smell candidates from our dataset, as shown in Step 2 of Figure 5.1. Specifically, we randomly chose one instance of each of the nine smell types across all 30 systems, resulting in an initial set of 270 candidates ($9 \times 30$). This set included both smelly and non-smelly classes and methods. During a careful review, we identified four instances of identical code with different names across two systems. To avoid duplication, we removed two of these instances, yielding

a final total of 268 code smell candidates for the human-centered ground truth. Step 3 of Figure 5.1 represents the evaluation performed by the participants. Participants rated the class code on a scale from 1 (it is definitely not a smell) to 5 (it is definitely a smell). It is important to note that at least two participants evaluated each code. In case of disagreement, a third participant evaluated the same code. The Human Ground Truth only includes code with an average score greater than 3.

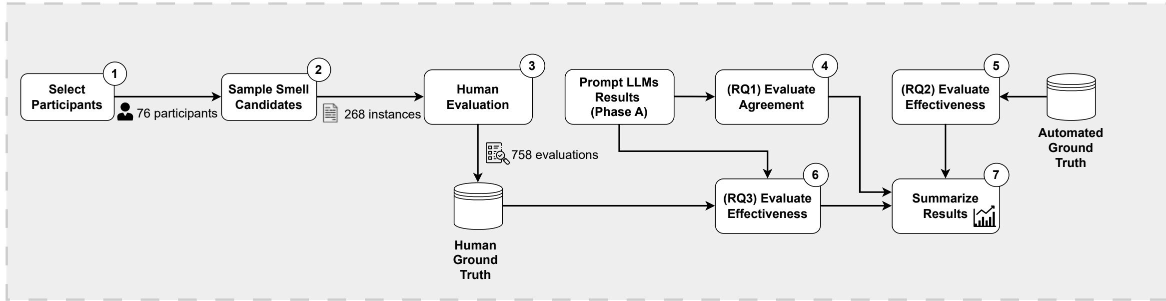



Figure 5.1: Steps of the Empirical Evaluation

Source: Elaborated by the author.

Steps 4 to 6 in Figure 5.1 indicate the analyzes we performed to answer each research question in this study. In Step 4, we analyze the agreement between the six automated detection approaches (that is, 4 LLMs and 2 static analysis tools) using Cohen's Kappa [41]. Steps 5 and 6 rely on the Automated Ground Truth and Human Ground Truth to evaluate the effectiveness of LLMs for code smell detection in terms of Precision, Recall, and F1-Score. In these steps, we also compare the effectiveness of LLMs with the static analysis tools used as baselines. Finally, we summarize our results in Step 7. In this step, we focus on the most interesting results and findings from our previous analyses. That is, we investigate the relations between agreement and effectiveness segregated by LLM and code smell.

## 5.3 Agreement between Code Smell Strategies

Table 5.1 presents the agreement analysis between the six automated strategies in the nine code smells considered in this study. The column "Second Tool" represents the other static analysis tool used to detect the respective code smell. For instance, for Data Class the first tool is Organic, and the second tool is PMD. Overall, the results reveal a heterogeneous landscape, with agreement levels ranging from poor (0.00) to moderate (0.66) depending on the smell and the pair of strategies compared. This variability

highlights the inherent difficulty of code smell detection and the differences in how static analysis tools and LLMs interpret code quality issues. Furthermore, the results also suggest that while some smells are consistently easier to detect across different approaches, others remain difficult regardless of the detection strategy employed.

Table 5.1: Analysis of Agreement between Different Tools and LLMs

| Smell | Tool | Second Tool | GPT-4o | Llama-3.3 | Qwen2.5-Coder | DeepSeek-R1 |
|---|---|---|---|---|---|---|
| Data Class | Organic | **0.66** | 0.18 | 0.48 | 0.08 | 0.12 |
| | PMD | - | 0.27 | 0.56 | 0.08 | 0.16 |
| | GPT-4o | - | - | 0.26 | 0.39 | 0.24 |
| | Llama-3.3 | - | - | - | 0.12 | 0.13 |
| | CodeQwen | - | - | - | - | 0.11 |
| Dispersed Coupling | Organic | **0.49** | 0.00 | 0.20 | 0.00 | 0.09 |
| | JSpIRIT | - | 0.00 | 0.13 | 0.00 | 0.09 |
| | GPT-4o | - | - | 0.00 | 0.00 | 0.00 |
| | Llama-3.3 | - | - | - | 0.00 | 0.09 |
| | CodeQwen | - | - | - | - | 0.00 |
| Feature Envy | Organic | **0.19** | 0.05 | 0.10 | 0.03 | 0.06 |
| | JDeodorant | - | 0.01 | 0.01 | 0.01 | 0.05 |
| | GPT-4o | - | - | 0.13 | 0.00 | 0.00 |
| | Llama-3.3 | - | - | - | 0.06 | 0.08 |
| | CodeQwen | - | - | - | - | 0.10 |
| Intensive Coupling | Organic | **0.53** | 0.02 | 0.15 | 0.01 | 0.10 |
| | JSpIRIT | - | 0.00 | 0.07 | 0.03 | 0.08 |
| | GPT-4o | - | - | 0.00 | 0.03 | 0.00 |
| | Llama-3.3 | - | - | - | 0.08 | 0.09 |
| | CodeQwen | - | - | - | - | 0.03 |
| Large Class | JSpIRIT | 0.49 | 0.33 | 0.23 | 0.00 | 0.27 |
| | JDeodorant | - | 0.34 | 0.26 | 0.00 | 0.26 |
| | GPT-4o | - | - | **0.60** | 0.00 | 0.24 |
| | Llama-3.3 | - | - | - | 0.00 | 0.16 |
| | CodeQwen | - | - | - | - | 0.00 |
| Long Method | Organic | **0.40** | 0.31 | 0.26 | 0.28 | 0.30 |
| | JDeodorant | - | 0.19 | 0.17 | 0.17 | 0.23 |
| | GPT-4o | - | - | 0.29 | 0.25 | 0.22 |
| | Llama-3.3 | - | - | - | 0.26 | 0.26 |
| | CodeQwen | - | - | - | - | 0.22 |
| Long Param. List | Organic | 0.08 | 0.02 | 0.27 | 0.11 | 0.15 |
| | PMD | - | **0.17** | 0.09 | 0.09 | 0.03 |
| | GPT-4o | - | - | 0.04 | 0.07 | 0.03 |
| | Llama-3.3 | - | - | - | 0.09 | 0.04 |
| | CodeQwen | - | - | - | - | 0.16 |
| Refused Bequest | Organic | **0.56** | 0.00 | 0.00 | 0.00 | 0.02 |
| | JSpIRIT | - | 0.00 | 0.00 | 0.00 | 0.02 |
| | GPT-4o | - | - | 0.00 | 1.00 | 0.00 |
| | Llama-3.3 | - | - | - | 0.00 | 0.00 |
| | CodeQwen | - | - | - | - | 0.00 |
| Shotgun Surgery | Organic | **0.38** | 0.00 | 0.02 | 0.00 | 0.01 |
| | JSpIRIT | - | 0.00 | 0.00 | -0.01 | 0.03 |
| | GPT-4o | - | - | 0.00 | 0.00 | 0.00 |
| | Llama-3.3 | - | - | - | 0.26 | 0.10 |
| | CodeQwen | - | - | - | - | 0.05 |

Static analysis tools, such as Organic and PMD, achieved moderate to good agree-

ment in detecting certain smells, such as Data Class (0.66), Refused Bequest (0.56) and Intensive Coupling (0.53), showing relative consistency between them. In contrast, LLMs showed much lower alignment, with Llama-3.3 occasionally performing better (e.g., 0.48 for Data Class, 0.20 for Dispersed Coupling, 0.26 for Shotgun Surgery), while GPT-4o, Qwen2.5-Coder, and DeepSeek-R1 generally scored close to zero. This result suggests that LLMs struggle to detect smells that require reasoning about inter-class dependencies and inheritance hierarchies [57], while static analysis heuristics are better suited to detect these smells. Overall, complex design smells represent a major challenge for LLMs.

Both static analysis tools and LLMs showed fair to moderate alignment on size-based smells, such as Large Class and Long Method, indicating that these smells are easier to detect consistently between different automated strategies. For Large Class, JSpIRIT and JDeodorant reached moderate agreement (0.49), while GPT-4o showed strong alignment with Llama-3.3 (0.60) and fair alignment with JDeodorant (0.34). Qwen2.5-Coder was the exception, showing zero alignment with all automated strategies. For Long Method, all tools and LLMs scored similarly (0.22–0.40), with GPT-4o, Llama-3.3, and Qwen2.5-Coder achieving 0.26–0.31. These results suggest that size-related smells are more consistently detected because they rely on clear structural metrics (e.g., lines of code, number of methods, cyclomatic complexity) and because they are widely supported and studied, allowing LLMs trained on large corpora to approximate traditional detection more effectively.

All automated detection strategies showed very low agreement for Feature Envy and Long Parameter List, confirming that these smells remain difficult to identify consistently. For Feature Envy, even static analysis tools, such as Organic and JDeodorant, achieved only poor agreement (0.19), as also reported by Oliveira et al. [76] using another dataset, and LLMs performed worse, with the best performance reaching only 0.10 (Llama-3.3). This difficulty might arise from excessive external reliance, a subjective and context-dependent distinction [57] that neither rule-based approaches nor current LLMs capture well. For Long Parameter List, the static analysis tools also reported poor agreement (0.08), while Llama-3.3 achieved the best LLM alignment at 0.27, with the others scoring lower. The inconsistent detection of this smell probably arises from varying thresholds of what constitutes “too many” parameters: static tools enforce rigid metrics [13], while LLMs reason more flexibly, increasing adaptability but reducing reproducibility.

**$RQ_1$ Findings:** Agreement between strategies is highly variable across code smells. Although size-based smells, such as Large Class and Long Method, show fair agreement between LLMs and static analysis tools, more complex smells, such as Feature Envy, Dispersed Coupling, Refused Bequest and Shotgun Surgery, exhibit poor or no agreement. These results suggest that LLMs are not yet reliable replacements for static analysis tools, particularly for design-level smells, although they show promise for simpler, size-based smells where simple context hints dominate.

## 5.4 Results with an Automated Ground Truth

Table 5.2 presents the effectiveness metrics (precision, recall, and F1-score) for each code smell, comparing static analysis tools and LLMs with the automatically generated ground truth. The columns "First Tool" and "Second Tool" represent the static analysis tools used to detect the respective code smells. The results show a wide variation across smells and strategies, with some smells being consistently easier to detect than others. In general, static tools tend to achieve higher precision, while LLMs often achieve higher recall, though at the cost of more false positives. This trade-off highlights the complementary nature of the two approaches: static tools are stricter and more conservative, detecting fewer smells but with higher accuracy, while LLMs cast a wider net, identifying more potential smells but also resulting in more false positives.

For Data Class, Feature Envy, and Intensive Coupling, static analysis tools showed very high precision but low recall, producing modest F1-scores overall. This indicates that static analysis tools are conservative, flagging few cases, but usually being correct when they do. LLMs, in contrast, achieved more balanced trade-offs by identifying more instances but also introducing false positives. GPT-4o achieved the most balanced detection for Data Class, while Llama-3.3 and Qwen2.5-Coder provided a more balanced detection of Feature Envy compared to static analysis tools. For Intensive Coupling, Qwen2.5-Coder and Llama-3.3 provided the closest results to static analysis tools. These results reveal that LLMs can complement static analysis tools for complex smells but remain less consistent.

For Dispersed Coupling, Refused Bequest, and Shotgun Surgery, all automated strategies performed poorly. Static analysis tools, while precise, achieved extremely low recall, and the LLMs struggled even more, often scoring close to zero. DeepSeek-R1 obtained the best LLM F1-score for Dispersed Coupling (0.30), but overall effectiveness

Table 5.2: Effectiveness Metrics for the Automated Ground Truth

| Smell (Tools) | Metrics | First Tool | Second Tool | GPT-4o | Llama-3.3 | Qwen2.5-Coder | DeepSeek-R1 |
|---|---|---|---|---|---|---|---|
| Data Class (Organic, PMD) | Precision | 0.98 | 0.94 | 0.82 | **1.00** | 0.33 | 0.49 |
| | Recall | 0.16 | 0.22 | **0.83** | 0.20 | 0.64 | 0.33 |
| | F1-Score | 0.28 | 0.36 | **0.82** | 0.34 | 0.43 | 0.39 |
| Dispersed Coupling (Organic, JSpIRIT) | Precision | **1.00** | **1.00** | 0.00 | **1.00** | 0.00 | **1.00** |
| | Recall | 0.06 | 0.06 | 0.00 | 0.10 | 0.00 | **0.18** |
| | F1-Score | 0.12 | 0.10 | 0.00 | 0.18 | 0.00 | **0.30** |
| Feature Envy (Organic, JDeodorant) | Precision | **1.00** | 0.77 | 0.37 | 0.77 | 0.47 | 0.70 |
| | Recall | 0.06 | 0.15 | **0.28** | 0.17 | 0.23 | 0.11 |
| | F1-Score | 0.11 | 0.25 | **0.32** | 0.28 | 0.31 | 0.19 |
| Intensive Coupling (Organic, JDeodorant) | Precision | 0.93 | 0.90 | 0.08 | 0.92 | 0.31 | **0.95** |
| | Recall | 0.14 | 0.20 | **0.33** | 0.16 | 0.29 | 0.11 |
| | F1-Score | 0.24 | **0.33** | 0.13 | 0.27 | 0.30 | 0.20 |
| Large Class (JSpIRIT, JDeodorant) | Precision | 0.86 | 0.91 | 0.99 | **1.00** | 0.01 | 0.55 |
| | Recall | 0.70 | 0.62 | 0.38 | 0.32 | 0.20 | **0.72** |
| | F1-Score | **0.77** | 0.74 | 0.55 | 0.48 | 0.02 | 0.63 |
| Long Method (Organic, JDeodorant) | Precision | 0.85 | 0.66 | 0.76 | **1.00** | 0.70 | 0.73 |
| | Recall | **0.84** | **0.84** | 0.78 | 0.61 | 0.79 | 0.80 |
| | F1-Score | **0.85** | 0.74 | 0.77 | 0.76 | 0.74 | 0.76 |
| Long Parameter List (Organic, PMD) | Precision | **0.99** | 0.44 | 0.21 | **0.99** | 0.82 | 0.89 |
| | Recall | 0.11 | 0.58 | **0.79** | 0.15 | 0.29 | 0.15 |
| | F1-Score | 0.20 | **0.50** | 0.33 | 0.26 | 0.43 | 0.25 |
| Refused Bequest (Organic, JSpIRIT) | Precision | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | Recall | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | F1-Score | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Shotgun Surgery (Organic, JSpIRIT) | Precision | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | Recall | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | F1-Score | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

remained limited. Both Refused Bequest and Shotgun Surgery had F1-scores of 0.00 across all tools and models. These results confirm that coupling-related and design-level smells are particularly difficult to detect, requiring reasoning about dependencies [57], inheritance hierarchies [34], and cross-cutting changes, which current tools and LLMs fail to capture effectively.

All strategies proved to be more consistent for Large Class, Long Method, and Long Parameter List. Size-based smells, such as Large Class and Long Method, achieved higher effectiveness, with static analysis tools reaching strong F1-scores (0.74–0.85) and LLMs performing competitively in some cases. Static analysis tools remained more effective considering the results for Long Parameter List, achieving the best F1-Score (PMD, 0.50), while all models scored between 0.25-0.43, with Qwen2.5-Coder providing the best F1-score. These findings show that both static analysis tools and LLMs handle size-based smells more reliably, while threshold-dependent smells, such as Long Parameter List, produce inconsistent results.

**$RQ_2$ Findings:** Static analysis tools generally achieve higher precision but lower recall, while LLMs detect more smells but introduce more false positives. For structural smells, such as Long Method and Large Class, both approaches perform well, with LLMs approaching tool effectiveness. However, for coupling-related and design-level smells (Dispersed Coupling, Feature Envy, Refused Bequest, Shotgun Surgery), all strategies show limited effectiveness. In general, LLMs are promising for structural smells, but remain unreliable for complex design issues.

## 5.5 Effectiveness of LLMs against Human Judgment

Table 5.3 compares the effectiveness of static analysis tools and LLMs with a human-based ground truth, following the same structure of Table 5.2. The columns "First Tool" and "Second Tool" represent the static analysis tools used to detect the respective code smells. Unlike the automated ground truth described in Section 5.4, this evaluation reflects how well automated strategies and developers align in the perception of code smells. The results show that for most smells (6 out 9), the static analysis tools achieve the best F1-score. Most LLMs achieved moderate to high recall, suggesting that they capture more of what humans consider smells, even if at the cost of false positives. This result makes human-based evaluation particularly important, as it highlights where automated approaches diverge from the way developers actually perceive code quality issues.

Table 5.3: Effectiveness Metrics for Human-based Ground Truth

| Smell (1º Tool, 2º Tool) | Metrics | First Tool | Second Tool | GPT-4o | Llama-3.3 | Qwen2.5-Coder | DeepSeek-R1 |
|---|---|---|---|---|---|---|---|
| Data Class (Organic, PMD) | Precision | 0.10 | 0.11 | 0.02 | **0.12** | 0.03 | 0.08 |
| | Recall | 0.40 | 0.55 | 0.23 | 0.38 | 0.44 | **0.59** |
| | F1-Score | 0.16 | **0.18** | 0.04 | **0.18** | 0.05 | 0.15 |
| Dispersed Coupling (Organic, JSpIRIT) | Precision | 0.17 | **0.40** | 0.00 | 0.16 | 0.00 | 0.10 |
| | Recall | **0.59** | 0.57 | 0.00 | 0.57 | 0.00 | 0.50 |
| | F1-Score | 0.27 | **0.47** | 0.00 | 0.25 | 0.00 | 0.16 |
| Feature Envy (Organic, JDeodorant) | Precision | **0.65** | 0.19 | 0.04 | 0.17 | 0.06 | 0.26 |
| | Recall | 0.62 | 0.52 | **0.67** | 0.65 | 0.64 | 0.60 |
| | F1-Score | **0.63** | 0.28 | 0.07 | 0.27 | 0.11 | 0.37 |
| Intensive Coupling (Organic, JDeodorant) | Precision | 0.32 | 0.26 | 0.01 | 0.28 | 0.06 | **0.48** |
| | Recall | **0.64** | 0.62 | 0.33 | 0.61 | 0.56 | 0.59 |
| | F1-Score | 0.43 | 0.36 | 0.02 | 0.38 | 0.10 | **0.53** |
| Large Class (JSpIRIT, JDeodorant) | Precision | 0.23 | 0.35 | 0.53 | **0.67** | 0.03 | 0.17 |
| | Recall | 0.58 | 0.61 | 0.59 | **0.73** | 0.67 | 0.46 |
| | F1-Score | 0.33 | 0.44 | 0.56 | **0.63** | 0.05 | 0.25 |
| Long Method (Organic, JDeodorant) | Precision | 0.46 | 0.33 | 0.46 | **0.85** | 0.47 | 0.50 |
| | Recall | **0.62** | 0.56 | 0.56 | 0.58 | **0.62** | 0.60 |
| | F1-Score | 0.53 | 0.42 | 0.51 | **0.69** | 0.54 | 0.55 |
| Long Parameter List (Organic, PMD) | Precision | **0.55** | 0.03 | 0.03 | 0.41 | 0.16 | 0.37 |
| | Recall | 0.61 | 0.50 | **1.00** | 0.59 | 0.53 | 0.56 |
| | F1-Score | **0.58** | 0.05 | 0.05 | 0.48 | 0.25 | 0.44 |
| Refused Bequest (Organic, JSpIRIT) | Precision | 0.13 | **0.19** | 0.00 | 0.00 | 0.00 | 0.01 |
| | Recall | 0.50 | **0.63** | 0.00 | 0.00 | 0.00 | 0.33 |
| | F1-Score | 0.20 | **0.29** | 0.00 | 0.00 | 0.00 | 0.02 |
| Shotgun Surgery (Organic, JSpIRIT) | Precision | 0.09 | **0.16** | 0.00 | 0.01 | 0.02 | 0.06 |
| | Recall | 0.61 | 0.61 | 0.00 | **1.00** | **1.00** | 0.53 |
| | F1-Score | 0.16 | **0.25** | 0.00 | 0.03 | 0.03 | 0.10 |

LLMs showed decent F1-scores for smells such as Intensive Coupling and Feature Envy. In Intensive Coupling, DeepSeek-R1 (0.53) outperformed both static tools (Organic 0.43, JDeodorant 0.36), with Llama-3.3 also performing well (0.38). In Feature Envy, although Organic achieved the highest F1-score (0.63), DeepSeek-R1 (0.37) surpassed JDeodorant (0.28). For Data Class and Dispersed Coupling, F1-scores were modest for all automated strategies, but PMD and Llama-3.3 tied (0.18) for Data Class, with DeepSeek-R1 (0.15) close behind. JSpIRIT achieved the overall best F1-score (0.47) for Dispersed Coupling. These results demonstrate that, while static analysis tools achieve better balance in Dispersed Coupling, LLMs can match or exceed at least one tool in

other complex smells, particularly when inter-class relationships must be reasoned about, suggesting real potential for these models when relational reasoning is involved.

Large Class, Long Method, and Long Parameter List formed the group of structural or size-related smells, where LLMs often rivaled or surpassed the F1-score results of static analysis tools. For Large Class, Llama-3.3 provided the strongest F1-score result on all detectors (0.63), followed by GPT-4o (0.56), both outperforming static tools (0.33 and 0.44). Similarly, models were highly competitive in Long Method, with Llama-3.3 again leading (0.69) and all LLMs (0.51–0.69) performing at least as well as tools (0.42–0.53). For Long Parameter List, Organic performed the best overall (0.58), but Llama-3.3 (0.48) and DeepSeek-R1 (0.44) were close behind, while PMD failed (0.05). These results show that for smells related to measurable structural properties, LLMs can reach or surpass tool effectiveness, although precision and recall balance continue to vary.

Refused Bequest and Shotgun Surgery remained the most difficult smells for all detectors, underlining their subjective and design-level complexity [57]. Static analysis tools performed poorly F1-scores, but better than LLMs for Refused Bequest (up to 0.29 with JSpIRIT), while all models remained near zero. Shotgun Surgery exhibited the same weakness: static analysis tools scored low but higher than most models (JSpIRIT 0.25 vs. GPT-4o 0.00, Llama-3.3 and Qwen2.5-Coder 0.03, DeepSeek-R1 0.10). Although Llama-3.3 and Qwen2.5-Coder recalled all true instances, their very low precision rendered their F1-scores negligible. These findings confirm that complex design-level smells remain extremely challenging for both automated strategies, requiring reasoning over inheritance and cross-class changes that neither static analysis tools nor these models effectively capture.

**$\text{RQ}_3$ Findings:** When compared to human judgment, static analysis tools generally remain more reliable, especially for Dispersed Coupling, Intensive Coupling, and Shotgun Surgery. However, LLMs show promise for structural smells, such as Large Class and Long Method, where models (Llama-3.3 and GPT-4o) even surpass static analysis tools. For Data Class and Feature Envy, LLMs achieve higher recall but very low precision, leading to many false positives. For Refused Bequest, all LLMs fail completely. In general, LLMs can complement tools by capturing more smells that humans perceive, but their low precision limits their standalone usefulness.

## 5.6 Threats to Validity

As with any empirical study, our evaluation of LLMs for code smell detection is subject to certain limitations that can affect the interpretation of the results. To ensure transparency and rigor, we discuss the main threats to validity that could influence our findings. Following established guidelines, we organize these threats into four categories: internal validity, external validity, construct validity, and conclusion validity [127]. Each category highlights potential risks and the measures that we have adopted to mitigate them.

**Internal validity** refers to elements that can unintentionally influence the results of a study without our knowledge [127]. An important threat lies in the configuration of the LLMs, such as temperature, output length, and prompt design. Small changes in these parameters could lead to different results, making it difficult to attribute the results solely to the inherent capabilities of the models. To mitigate this threat, we standardized the prompt in all LLMs, fixed the temperature values according to model recommendations, and documented all settings in our replication package [1]. Another internal threat is the potential bias introduced by human participants in manual evaluation. Although we provided training and ensured that each instance was reviewed by at least two participants (with a third in case of disagreement), individual differences in experience and interpretation of smells may still have influenced the results. Finally, the imbalance in the dataset, with some smells much more frequent than others, may have affected the F1-scores, favoring automated strategies that perform well on the most common smells.

**External validity** refers to the extent to which our findings can be generalized beyond the specific context of this study [127]. A key limitation is that our evaluation focused on Java systems, which may not reflect how LLMs and tools perform in other programming languages. Smells in dynamic languages, such as JavaScript or Python, for example, may present different challenges. Another limitation is the selection of LLMs: although we included four representative models (GPT-4o, Llama-3.3, Qwen2.5-Coder, and DeepSeek-R1), the landscape of LLMs evolves rapidly, and newer models or fine-tuned variants may yield different results. Similarly, the static analysis tools we used represent widely adopted baselines, but other tools with different strategies could produce different outcomes. Finally, human evaluation was conducted with undergraduate students, who may not fully represent the perceptions of professional developers regarding code smells. These factors limit the generalization of our conclusions, although the study design allows for replication in broader contexts.

**Construct validity** relates to whether the measures we used accurately capture

[1] https://zenodo.org/records/16790193

the concepts under investigation [127]. A significant threat is the way we define the ground truths. The automated ground truth relied on the majority voting between static analysis tools and LLMs, which may have introduced bias if both types of detectors systematically missed or misclassified certain smells. Similarly, the human ground truth was based on majority voting from students, which may not perfectly reflect expert consensus. Another construct threat is the interpretation of effectiveness metrics. Precision, recall, and F1-score provide useful insights, but they do not capture all aspects of detection quality, such as the severity of false positives or the practical usefulness of flagged smells. Finally, treating missing or malformed LLM outputs as negative predictions may have underestimated their potential effectiveness. To mitigate these threats, we followed established practices from previous studies, reported all methodological decisions transparently, and provided replication material to allow others to test alternative definitions and metrics.

## 5.7 Chapter Summary

In this chapter, we evaluated the effectiveness of LLMs for code smell detection by comparing them with static analysis tools and human judgment. We first examined the level of agreement between detectors, finding that LLMs and tools align moderately well for structural smells, such as Large Class and Long Method, but show poor or no agreement for more complex design smells, such as Feature Envy, Refused Bequest, and Shotgun Surgery. We then assessed effectiveness against an automated ground truth, where static tools generally achieved higher precision while LLMs achieved higher recall, with GPT-4o standing out for Data Class and all detectors performing well for Long Method. Finally, we compared detectors with a human-based ground truth, which revealed that static tools remain more reliable for coupling-related and design-level smells, while LLMs, particularly Llama-3.3 and GPT-4o, showed strong performance for structural smells, such as Large Class and Long Method. In general, our results indicate that LLMs can complement traditional tools by capturing more instances of structural smells, but their low precision and limited effectiveness for complex code smells prevent them from serving as standalone solutions.

Building on these findings, the next chapter concludes this dissertation by summarizing the main contributions of our work and discussing its implications for both researchers and practitioners. In addition, the chapter outlines future research opportunities, setting the stage for continued progress in automated code smell detection.

# Chapter 6

# Conclusion

Code smells are indicators of design or implementation issues that, while not necessarily bugs, can hinder software maintainability, readability, and evolution if left unaddressed [43, 79, 94, 95, 104, 130]. Detecting these smells early is essential to reduce technical debt and improve long-term software quality. Traditional static analysis tools provide support for this task, but often struggle with more complex design issues [28, 29, 103]. In this context, LLMs have emerged as a promising alternative, given their ability to capture semantic and contextual information from source code. This dissertation investigated whether LLMs can effectively support code smell detection, showing that they complement static analysis tools by performing well on some structural smells, such as Large Class and Long Method, while still facing challenges with design-level smells, such as Feature Envy, Refused Bequest, and Shotgun Surgery. Our contributions include a Systematic Literature Review (SLR) on automated detection strategies, the construction of an extended dataset with an automated ground truth, and an empirical evaluation comparing LLMs, static analysis tools, and human judgment. These findings highlight both the potential and the current limitations of LLMs and point to future research opportunities in hybrid detection strategies, improved datasets, and framework-specific analyses.

The remainder of this chapter is organized as follows. Section 6.1 provides an overview of the work conducted in this dissertation. Section 6.2 summarizes the main contributions of our study. Section 6.3 discusses the implications of our findings for researchers and practitioners. Finally, Section 6.4 outlines some directions for future work, highlighting opportunities to advance the use of LLMs in automated code smell detection.

## 6.1 Work Overview

This dissertation investigated the state of the art in automated code smell detection (Chapter 3). Through a SLR, we identified 22 automated detection strategies and analyzed their features, detection techniques, and evolution over time. The results showed that most strategies rely on rule-based linting and target relatively simple structural smells, such as Long Method and Large Class, while more complex design-level smells, such as Feature Envy or Shotgun Surgery, remain underexplored. We also observed that industry-driven tools, such as ESLint and JSLint, dominate the JavaScript ecosystem and continue to receive active maintenance, while research-driven tools often lack long-term support. This review highlighted important research gaps, including the limited adoption of AI-based techniques, motivating the exploration of LLMs as a new approach to code smell detection.

Based on these findings, Chapter 4 introduced the dataset used in our empirical evaluation. We extended an existing dataset of 3,459 code smell instances from 30 popular Java projects [94] incorporating LLM-based detection results. To ensure consistency, we created an automated ground truth using a majority voting strategy that combined outputs from static analysis tools and LLMs. We used a zero-shot prompt to simulate realistic developer interactions with LLMs and selected four representative models (GPT-4o, Llama-3.3, Qwen2.5-Coder, and DeepSeek-R1) for evaluation. This dataset provided the foundation for the empirical study in Chapter 5, allowing a systematic comparison between LLMs, static analysis tools, and human judgment.

In Chapter 5, we evaluated the effectiveness of LLMs for code smell detection through three complementary perspectives. First, we analyzed the agreement between detectors and found that LLMs and static analysis tools align moderately well for structural smells, such as Large Class and Long Method, but show poor or no agreement for design-level smells, such as Feature Envy, Refused Bequest, and Shotgun Surgery. Second, we assessed the effectiveness against an automated ground truth, where static analysis tools generally achieved higher precision, while LLMs achieved higher recall. Finally, we compared detectors with a human-based ground truth, which revealed that static analysis tools remain more reliable for coupling-related and design-level smells, while LLMs performed strongly for structural smells, in some cases surpassing static analysis tools. Together, these results demonstrate that LLMs can complement traditional tools by broadening coverage of structural smells, but their low precision and limited effectiveness for complex design smells prevent them from serving as standalone solutions.

## 6.2 Main Contributions

Our main contributions from this work include:

1. A SLR of automated strategies in detecting code smells for JavaScript systems, describing each strategy in terms of their availability, supported smells, among other features;
2. An automated ground truth for code smell detection, obtained through a majority voting process that combines results from static analysis tools and LLMs;
3. An empirical study comparing LLMs with static analysis tools in code smell detection, evaluating their agreement and effectiveness using precision, recall, and F1-score across nine types of code smells;
4. The creation of a human-validated ground truth and a comparative analysis to examine how closely LLMs and tools reflect the perception of developers about code smells;
5. Publicly available replication packages for Study 1 [1], Study 2 [2] and Study 3 [2];

Contribution 1 addresses the gap identified in Research Problem 1 by consolidating fragmented knowledge on automated code smell detection strategies. This catalog, produced in Study 1 (Chapter 3), organizes strategies by availability, supported smells, and technical features, offering both researchers and practitioners a structured overview of the field. Contribution 2 responds to the limitation described in Research Problem 2 by introducing an automated ground truth for code smell detection. Built in Study 2 (Chapter 4) through a majority vote between static analysis tools and LLMs, this dataset provides a reproducible and extensible benchmark that can be used to evaluate current and future detection approaches.

Contributions 3 and 4 jointly address the challenges raised in Research Problems 3 and 4 by providing a comprehensive empirical evaluation of LLMs compared to static analysis tools and human judgment. The evaluation performed in Study 3 (Chapter 5) measured agreement, precision, recall, and F1-score in nine code smells, while also incorporating a human-validated ground truth to assess how well automated approaches align with the developer perception. Together, these contributions fill a critical gap in the literature by clarifying both the relative performance of LLMs and their alignment with human expectations. Contribution 5 complements the previous contributions by releasing replication packages for Studies 1, 2, and 3. These packages ensure transparency,

[1] https://zenodo.org/records/15757269
[2] https://zenodo.org/records/16790193

reproducibility, and extensibility, enabling other researchers to replicate our findings or extend studies with new tools, models, or human evaluations.

## 6.3 Study Implications

The findings of this dissertation have several implications for software practitioners who rely on automated strategies to maintain code quality. From the SLR in Chapter 3, we observed that most existing JavaScript tools focus on simple structural smells, such as Long Method and Large Class, while neglecting more complex design-level issues. This suggests that practitioners should be cautious when relying solely on static analysis tools, as they may overlook deeper architectural problems. The empirical results in Chapter 5 further reinforce this point: static analysis tools remain highly precise but conservative, often missing smells that developers would identify, while LLMs broaden coverage by detecting more potential smells, albeit with more false positives. For practitioners, this means that LLMs can serve as valuable complements to existing tools, particularly for surfacing structural smells that may otherwise go unnoticed. However, their outputs should be carefully reviewed, ideally in combination with results from static analysis tools, to avoid unnecessary refactorings. Integrating LLMs into code review pipelines or IDEs could provide developers with richer feedback, but adoption should be gradual and supported by human oversight.

For researchers, the results highlight both opportunities and challenges in advancing automated code smell detection. Chapter 3 revealed a limited adoption of AI-based techniques for code smell detection and a lack of benchmark datasets that include the output of LLMs. Chapter 4 addressed this gap by extending an existing dataset with LLM-based outputs and building an automated ground truth through majority voting. This dataset, together with the human-based ground truth created in Chapter 5, provides a valuable resource for future studies. Researchers can use these datasets to benchmark new detection techniques, compare emerging LLMs, or explore hybrid approaches that combine static analysis with AI-driven reasoning. The empirical evaluation in Chapter 5 also showed that LLMs perform well on structural smells but struggle with design-level ones, suggesting that future research should focus on improving prompt engineering, fine-tuning models on code smell datasets, and developing hybrid detection strategies that take advantage of both metrics and semantic reasoning.

The datasets produced in this dissertation can serve as a basis for both applied and academic work. Practitioners can use them to assess the effectiveness of tools in real-world scenarios, while researchers can extend them by incorporating additional programming

languages, frameworks, or smell types. For example, future work could expand the dataset to include framework-specific smells in JavaScript (e.g., React or Vue.js), or to cover other ecosystems, such as Python or C#. The automated ground truth can also be updated as new LLMs and tools emerge, ensuring that the benchmark remains relevant. Furthermore, the human-based ground truth provides a unique opportunity to study how developers perceive smells, which can inform the design of more developer-aligned detection strategies. By making these datasets available and extensible, this dissertation contributes not only to the evaluation of current tools and LLMs but also to the long-term development of more reliable and context-aware approaches to code smell detection.

## 6.4 Future Work

For future work, we propose a focus on advancing detection techniques that go beyond the limitations of both static analysis tools and current LLMs. Our results showed that LLMs perform well in structural smells, but struggle with design-level issues such as Feature Envy, Refused Bequest, and Shotgun Surgery. This result opens opportunities for exploring hybrid approaches that combine the precision of metrics with the contextual reasoning of LLMs. Fine-tuning LLMs on curated code smell datasets, experimenting with advanced prompting strategies, such as chain-of-thought or few-shot learning, and integrating architectural information (e.g., dependency graphs or change history) could improve their ability to capture complex smells. Another promising direction is the use of advanced prompting strategies that not only detect smells but also provide interpretable justifications and actionable refactoring suggestions, which would increase developer trust and adoption.

In addition, future work could expand and refine the datasets introduced in this dissertation. Extending both automated and human-based ground truths to include other programming languages, frameworks, and smell types would improve generalization and enable cross-language comparisons. For example, incorporating framework-specific smells in JavaScript (e.g., React or Angular) or smells in other languages, such as Python, could reveal new challenges and opportunities. More diverse human evaluations, involving professional developers with varying levels of expertise, would also strengthen the reliability of the ground truths. Finally, studies that integrate LLM-based detectors into real development workflows could provide information on their practical impact on software quality, technical debt reduction, and developer productivity. By addressing these directions, future research can move closer to building robust, adaptive, and developer-aligned solutions for automated code smell detection.

# Bibliography


[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190, 2011.

[2] Naser Al Madi. How readable is model-generated code? examining readability and visual inspection of github copilot. In *International Conference on Automated Software Engineering (ASE)*, pages 1–5, 2022.

[3] Hamoud Aljamaan. Voting heterogeneous ensemble for code smell detection. In *20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 897–902, 2021.

[4] Nabil Almashfi and Lunjin Lu. Code smell detection tool for java script programs. In *2020 5th International Conference on Computer and Communication Systems (ICCCS)*, pages 172–176, 2020.

[5] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. Automatic Library Migration Using Large Language Models: First Results. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–7, 2024.

[6] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*, pages 185–196, 2024.

[7] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Márcio Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 261–269, 2015.

[8] Hirojiro Aoyama. A study of stratified random sampling. *Ann. Inst. Stat. Math*, 6(1):1–36, 1954.

[9] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering (EMSE)*, 21:1143–1191, 2016.

[10] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology (IST)*, 108:115–138, 2019.

[11] Aryclenio Barros and Eiji Adachi. Bad smells in javascript: A mapping study. In *Proceedings of the 9th Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pages 1–5, 2021.

[12] Victor Basili and H Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering (TSE)*, 14(6):758–773, 1988.

[13] Anushka Bhave and Roopak Sinha. Deep multimodal architecture for detection of long parameter list and switch statements using distilbert. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–120, 2022.

[14] Justus Bogner and Manuel Merkel. To type or not to type? a systematic comparison of the software quality of javascript and typescript applications on github. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, pages 658–669, 2022.

[15] Uriel Ferreira Campos, Guilherme Smethurst, João Pedro Moraes, Rodrigo Bonifácio, and Gustavo Pinto. Mining rule violations in javascript code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 195–199, 2019.

[16] Joanne Carneiro, Jessica Barbara da Silva Ribas, Amanda Santana, Eduardo Figueiredo, and Juliana Alves Pereira. Improvmlcq: A feature-enriched dataset or advancing code smell detection. In *Ibero-American Conference on Software Engineering (CIbSE)*, pages 165–179, 2025.

[17] Genevieve Caumartin, Qiaolin Qin, Sharon Chatragadda, Janmitsinh Panjrolia, Heng Li, and Diego Elias Costa. Exploring the potential of llama models in automated code refinement: A replication study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 681–692, 2025.

[18] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. ChatUniTest: A Framework for LLM-Based Test Generation. In *International Conference on the Foundations of Software Engineering (FSE)*, 2024.

[19] Jinsu Choi, Gabin An, and Shin Yoo. Iterative refactoring of real-world open-source programs with large language models. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 49–55, 2024.

[20] Daniel Cruz, Amanda Santana, and Eduardo Figueiredo. Detecting bad smells with machine learning algorithms: an empirical study. In *Proceedings of the 3rd International Conference on Technical Debt (TechDebt)*, pages 31–40, 2020.

[21] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software (JSS)*, 203:111734, 2023.

[22] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. A systematic literature review on bad smells–5 w's: which, when, what, who, where. *IEEE Transactions on Software Engineering (TSE)*, 47(1):17–66, 2018.

[23] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[24] Chunhao Dong, Yanjie Jiang, Yuxia Zhang, Yang Zhang, and Liu Hui. Chatgpt-based test generation for refactoring engines enhanced by feature analysis on examples. In *International Conference on Software Engineering (ICSE)*, pages 746–746, 2025.

[25] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 33(7):1–38, 2024.

[26] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13, 2024.

[27] Mateus Dutra, Denis Pinheiro, Johnatan Oliveira, and Eduardo Figueiredo. From detection to refactoring of microservice bad smells: A systematic literature review. *Journal of Software Engineering (JSE)*, 6:1, 2019.

[28] Amin Milani Fard and Ali Mesbah. Jsnose: Detecting javascript code smells. In *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

[29] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–12, 2016.

[30] Fabio Ferreira and Marco Tulio Valente. Detecting code smells in react-based web apps. *Information and Software Technology (IST)*, 155:107111, 2023.

[31] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039, 2011.

[32] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[33] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mäntylä. Code smell detection: Towards a machine learning-based approach. In *International Conference on Software Maintenance (ICSM)*, pages 396–399, 2013.

[34] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[35] Tommaso Fulcini, Giacomo Garaccione, Riccardo Coppola, Luca Ardito, and Marco Torchiano. Guidelines for gui testing maintenance: a linter for test smell detection. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST)*, pages 17–24, 2022.

[36] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1–13, 2024.

[37] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. Dlint: Dynamically checking bad coding practices in javascript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 94–105, 2015.

[38] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *European Conference on Information Retrieval (ECIR)*, pages 345–359, 2005.

[39] Salvatore Guarnieri. {GULFSTREAM}: Staged static analysis for streaming {JavaScript} applications. In *USENIX Conference on Web Application Development (WebApps)*, 2010.

[40] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. Exploring the potential of chatgpt in automated code refinement:

An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 1–13, 2024.

[41] Kilem L Gwet. *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters.* Advanced Analytics, LLC, 2014.

[42] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Arpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101, 2019.

[43] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–39, 2014.

[44] Jayant Havare, Varsha Apte, Kaushikraj Maharajan, Nithin Chandra Gupta Samudrala, Ganesh Ramakrishnan, Srikanth Tamilselvam, and Sainath Vavilapalli. Ai-based automated grading of source code of introductory programming assignments. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*, pages 171–181, 2025.

[45] Tjaša Heričko and Boštjan Šumak. Analyzing linter usage and warnings through mining software repositories: A longitudinal case study of javascript packages. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1375–1380, 2022.

[46] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 33(8):1–79, 2024.

[47] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology (IST)*, 93:130–146, 2018.

[48] Xiaoping Jia and Howard Dittmer. Anomaly detection in dynamic programming languages through heuristics based type inference. In *2017 Computing Conference*, pages 286–293, 2017.

[49] Jinan Jiang, Zihao Li, Haoran Qin, Muhui Jiang, Xiapu Luo, Xiaoming Wu, Haoyu Wang, Yutian Tang, Chenxiong Qian, and Ting Chen. Unearthing gas-wasting code smells in smart contracts with large language models. *IEEE Transactions on Software Engineering (TSE)*, 2024.

[50] David Johannes, Foutse Khomh, and Giuliano Antoniol. A large-scale empirical study of code smells in javascript projects. *Software Quality Journal (SQJ)*, 27:1271–1314, 2019.

[51] Dalton Jorge, Patricia Machado, and Wilkerson Andrade. Investigating test smells in javascript test code. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing (SAST)*, pages 36–45, 2021.

[52] Junaed Younus Khan and Gias Uddin. Automatic code documentation generation using gpt-3. In *International Conference on Automated Software Engineering (ASE)*, pages 1–6, 2022.

[53] Vaishali Khanve. Are existing code smells relevant in web games? an empirical study. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1241–1243, 2019.

[54] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *2009 Ninth International Conference on Quality Software (QSIC)*, pages 305–314, 2009.

[55] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University and University of Durham, 2007.

[56] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIWare)*, pages 103–111, 2024.

[57] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2006.

[58] Yi Liu. Jsoptimizer: an extensible framework for javascript program optimization. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 168–170, 2019.

[59] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering (TSE)*, 50(6):1548–1584, 2024.

[60] Jefferson GM Lopes, Johnatan Oliveira, and Eduardo Figueiredo. Evaluating the impact of developer experience on code quality: A systematic literature review. In *Ibero-American Conference on Software Engineering (CIbSE)*, pages 166–180, 2024.

[61] Mateus Lopes and Andre Hora. How and why we end up with complex methods: a multi-language study. *Empirical Software Engineering (EMSE)*, 27(5):115, 2022.

[62] Lech Madeyski and Tomasz Lewowski. Mlcq: Industry-relevant code smell data set. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 342–347, 2020.

[63] Lech Madeyski and Tomasz Lewowski. Detecting code smells using industry-relevant data. *Information and Software Technology (IST)*, 155:107112, 2023.

[64] Zhelu Mai, Jinran Zhang, Zhuoer Xu, and Zhaomin Xiao. Financial sentiment analysis meets llama 3: A comprehensive analysis. In *Proceedings of the 2024 7th International Conference on Machine Learning and Machine Intelligence (MLMI)*, pages 171–175, 2024.

[65] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aimeur. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 278–281, 2012.

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

[67] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. On the robustness of code generation techniques: An empirical study on github copilot. In *International Conference on Software Engineering (ICSE)*, pages 2149–2160, 2023.

[68] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)*, 36(1):20–36, 2009.

[69] Robert C Moore and William Lewis. Intelligent selection of language model training data. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 220–224, 2010.

[70] Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS)*, pages 5–14, 2010.

[71] Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Software*, 41(1):105–112, 2023.

[72] Henrique Gomes Nunes, Amanda Santana, Eduardo Figueiredo, and Heitor Costa. Tuning code smell prediction models: A replication study. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 316–327, 2024.

[73] David OBrien, Sumon Biswas, Sayem Mohammad Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan. Are prompt engineering and todo comments friends or foes? an evaluation on github copilot. In *International Conference on Software Engineering (ICSE)*, pages 1–13, 2024.

[74] Frolin S Ocariza, Karthik Pattabiraman, and Ali Mesbah. Detecting unknown inconsistencies in web applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 566–577, 2017.

[75] Fernando L Oliveira and Júlio CB Mattos. Jsguide: A tool to improve javascript algorithms focusing on iot devices. In *Symposium on Internet of Things (SIoT)*, pages 1–4, 2022.

[76] Igor Oliveira, Joanne Carneiro, Jessica Ribas, and Juliana Pereira. Code smell classification in python: Are small language models up to the task? In *Brazilian Symposium on Software Engineering (SBES)*, pages 699–705, 2025.

[77] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development (JSERD)*, 5:1–28, 2017.

[78] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, 2013.

[79] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. In *International Conference on Software Engineering (ICSE)*, pages 482–482, 2018.

[80] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering (TSE)*, 41(5):462–489, 2014.

[81] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. A textual-based technique for smell detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.

[82] Aikaterini Paltoglou, Vassilis E Zafeiris, Emmanouel A Giakoumakis, and NA Diamantidis. Automated refactoring of client-side javascript code to es6 modules. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 402–412, 2018.

[83] Katerina Paltoglou, Vassilis E Zafeiris, NA Diamantidis, and Emmanouel A Giakoumakis. Automated refactoring of legacy javascript code to es6 modules. *Journal of Systems and Software (JSS)*, 181:111049, 2021.

[84] Chansol Park and R Young Chul Kim. Detecting common weakness enumeration through training the core building blocks of similar languages based on the codebert model. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 641–642, 2023.

[85] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of javascript web applications in the wild. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE-Companion)*, pages 61–70, 2016.

[86] Jorge Pérez, Jessica Díaz, Javier Garcia-Martin, and Bernardo Tabuenca. Systematic literature reviews in software engineering—enhancement of the study selection process using cohen's kappa statistic. *Journal of Systems and Software (JSS)*, 168:110657, 2020.

[87] PMD. Pmd source code analyzer, 2025. https://pmd.github.io/.

[88] Chanathip Pornprasit and Chakkrit Tantithamthavorn. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology (IST)*, 175:107523, 2024.

[89] Adam A Porter, Lawrence G Votta, and Victor R Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering (TSE)*, 21(6):563–575, 2002.

[90] José Pereira dos Reis, Fernando Brito e Abreu, and Glauco de Figueiredo Carneiro. Crowdsmelling: A preliminary study on using collective knowledge in code smells detection. *Empirical Software Engineering (EMSE)*, 27(3):69, 2022.

[91] Américo Rio, Fernando Brito e Abreu, and Diana Mendes. Causal inference of server-and client-side code smells in web apps evolution. *Empirical Software Engineering (EMSE)*, 29(5):133, 2024.

[92] Zhyar Rzgar K Rostam, Sándor Szénási, and Gábor Kertész. Achieving peak performance for large language models: A systematic review. *IEEE access*, 2024.

[93] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. An empirical study of code smells in javascript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305, 2017.

[94] Amanda Santana, Eduardo Figueiredo, and Juliana Alves Pereira. Unraveling the impact of code smell agglomerations on code stability. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 461–473, 2024.

[95] Geanderson Santos, Amanda Santana, Gustavo Vale, and Eduardo Figueiredo. Yet another model! a study on model's similarities for defect and code smells. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 282–305, 2023.

[96] José Amancio M Santos, João B Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos Do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes De Mendonça. A systematic review on the code smell effect. *Journal of Systems and Software (JSS)*, 144:450–477, 2018.

[97] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering (TSE)*, 50(1):85–105, 2023.

[98] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2010.

[99] Oussama Ben Sghaier, Martin Weyssow, and Houari Sahraoui. Harnessing large language models for curated code reviews. In *International Conference on Mining Software Repositories (MSR)*, pages 187–198, 2025.

[100] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software (JSS)*, 176:110936, 2021.

[101] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. Designite: A software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities (BRIDGE)*, pages 1–4, 2016.

[102] Ian Shoenberger, Mohamed Wiem Mkaouer, and Marouane Kessentini. On the use of smelly examples to detect code smells in javascript. In *European Conference*

*on the Applications of Evolutionary Computation (EvoApplications)*, pages 20–34, 2017.

[103] Luciana Lourdes Silva, Janio Rosa da Silva, João Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. Detecting code smells using chatgpt: Initial insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 400–406, 2024.

[104] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering (TSE)*, 39(8):1144–1156, 2012.

[105] Abdel Aziz Taha and Allan Hanbury. Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. *BMC Medical Imaging*, 15(1):29, 2015.

[106] Damian A Tamburri, Fabio Palomba, Alexander Serebrenik, and Andy Zaidman. Discovering community patterns in open-source: a systematic approach and its evaluation. *Empirical Software Engineering (EMSE)*, 24:1369–1417, 2019.

[107] Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software (JSS)*, 199:111623, 2023.

[108] Minaoar Hossain Tanzil, Junaed Younus Khan, and Gias Uddin. Chatgpt incorrectness detection in software reviews. In *International Conference on Software Engineering (ICSE)*, pages 1–12, 2024.

[109] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345, 2010.

[110] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering (TSE)*, 39(10):1427–1443, 2013.

[111] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering (TSE)*, 46(8):863–891, 2018.

[112] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM Sigplan Notices*, 34(10):47–56, 1999.

[113] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–331, 2008.

[114] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering (TSE)*, 43(11):1063–1088, 2017.

[115] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *International Conference on Software Engineering (ICSE)*, pages 2291–2302, 2022.

[116] Alexi Turcotte, Michael D Shah, Mark W Aldrich, and Frank Tip. Drasync: identifying and visualizing anti-patterns in asynchronous javascript. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 774–785, 2022.

[117] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering (WCRE)*, pages 97–106, 2002.

[118] Natthida Vatanapakorn, Chitsutha Soomlek, and Pusadee Seresangtakul. Python code smell detection using machine learning. In *2022 26th International Computer Science and Engineering Conference (ICSEC)*, pages 128–133, 2022.

[119] Kuldeep Vayadande, Kuhu Mukhopadhyay, Vaishnavi Chaudhari, Shreyas Manwadkar, Tanmay Mutalik, and Ishan Gawali. Let us lint: A tool for code formatting and code enhancing. In *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–8, 2023.

[120] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6, 2015.

[121] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-based Code Completion. In *International Conference on Software Engineering (ICSE)*, pages 781–781, 2025.

[122] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering (TSE)*, 2024.

[123] Wei Wang, Vincent W Zheng, Han Yu, and Chunyan Miao. A survey of zero-shot learning: Settings, methods, and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–37, 2019.

[124] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271, 2020.

[125] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:24824–24837, 2022.

[126] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–10, 2014.

[127] Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering (ESE)*. Springer Science & Business Media, 1st edition, 2012.

[128] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. ismell: Assembling llms with expert toolsets for code smell detection and refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1345–1357, 2024.

[129] Yuankai Xue, Hanlin Chen, Gina R Bai, Robert Tairas, and Yu Huang. Does chatgpt help with introductory programming? an experiment of students using chatgpt in cs1. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pages 331–341, 2024.

[130] Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software (JSS)*, 86(10):2639–2653, 2013.

[131] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, 2013.

[132] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering (PACMSE)*, 1:1703–1726, 2024.

[133] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397, 2020.

[134] Jinran Zhang, Zhelu Mai, Zhuoer Xu, and Zhaomin Xiao. Is llama 3 good at identifying emotion? a comprehensive study. In *Proceedings of the 2024 7th International Conference on Machine Learning and Machine Intelligence (MLMI)*, pages 128–132, 2024.

[135] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 5673–5684, 2023.

[136] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. Migrating Code At Scale With LLMs At Google. In *International Conference on the Foundations of Software Engineering (FSE)*, pages 1–12, 2025.