SBC Reviews on Computer Science, 2025, 5:1, doi:

A Systematic Literature Review of Code Smell Detection Tools for JavaScript Systems

Saymon Souza [**Federal University of Minas Gerais** | ***silvasouzasaymon@dcc.ufmg.br***]
Felipe Ribeiro [**Federal University of Minas Gerais** | ***felipe.ribeiro@dcc.ufmg.br***]
Eduardo Fernandes [**SENAI-SP University Center** | ***emfernandes@acm.org***]
Eduardo Figueiredo [**Federal University of Minas Gerais** | ***figueiredo@dcc.ufmg.br***]

Abstract JavaScript is one of the most popular programming languages. As projects grow, their code can become complex which leads to code smells, signs that can indicate problems. Many tools are available to detect and fix these issues, but we need a comprehensive summary of their most important features. This paper provides a systematic literature review of JavaScript code smell detection tools. We searched four databases (Scopus, ACM Digital Library, IEEE Xplore, and Springer) using a specific search string to find relevant studies. To refine the results, we applied a four-step selection process, reducing 1002 initial studies to 18 by removing duplicates, filtering metadata, and reviewing their full texts. We then used backward and forward snowballing to find more relevant studies, increasing the total number to 27 primary studies. Finally, we examined these studies to analyze the code smell detection tools they described. We identified 22 tools, many published in top software engineering venues, such as ICSE, MSR and TSE. We found that most tools use rule-based linting (55%), which is efficient but struggles with complex architectural smells. Dynamic analysis (23%) is underused and AI-driven detection is completely missing, despite its relevance in modern software engineering research. Researchers are also developing framework-specific tools for modern JavaScript practices and focusing on the detection of test smells (22%). Most tools available to practitioners detect only basic smells and ignore deeper design issues. Tool builders can address these gaps by combining static and dynamic analysis and creating more adaptable tools. For researchers, the lack of AI-driven detection and modern benchmark datasets presents an opportunity for progress.

Keywords: JavaScript Source Code, Code Smell Detection, Systematic Literature Review, Maintainability, Software design

1 Introduction

JavaScript is a dynamically typed programming language that remains highly popular year after year (Andreasen et al., 2017). The distinctive features of JavaScript pose specific challenges for code smell detection that differ significantly from those of statically typed, class-based languages such as Java (Wei and Ryder, 2013). Its dynamic typing and prototype-based inheritance make it difficult to apply traditional static analysis techniques, as objects and their structures can change at runtime, undermining the assumptions made by static analyzers (Jensen et al., 2011). These characteristics lead to quality issues that are less prevalent or easier to detect in statically typed languages (Rastogi et al., 2015).

In fact, the rapid evolution of frameworks, such as React, Angular, and Vue.js, shapes modern JavaScript development, which introduces new architectural patterns and anti-patterns not typically found in other ecosystems (Ferreira and Valente, 2023). The prevalence of asynchronous programming and higher-order functions further increases the complexity of detecting smells, such as callback hell, missing error handling, or deeply nested logic (Turcotte et al., 2022). These aspects highlight the need for a tailored analysis. That is, developers and researchers should reason about traditional smells under the semantics of JavaScript and new categories of smells could also emerge, which require tools and taxonomies designed with these unique traits in mind.

Fowler (1999) first introduced the term “code smell”, and since then researchers have widely studied this concept. Although researchers originally applied this concept to code that breaks object-oriented principles, subsequent work expanded it to include more languages and programming paradigms (Santos et al., 2018). The results of the work performed by Marinescu (2004) show that files with code smells are more likely to have maintenance problems, highlighting the need to detect and fix these issues to improve overall software quality. Manual analysis can yield satisfactory results, but this may require significant effort from the development team. Therefore, tool developers have increasingly created automated tools to support development teams prevent and fix code smells with less manual effort (Fokaefs et al., 2011; Van Emden and Moonen, 2002; Vidal et al., 2015).

Despite the popularity of JavaScript and the growing number of code smell detection tools for this language, the current body of research remains fragmented and methodologically inconsistent (Barros and Adachi, 2021). This suggests an uneven coverage of code quality concerns, which may lead to tool misalignment with real-world development challenges (Di Nucci et al., 2018). Moreover, while other programming ecosystems have begun adopting AI-based techniques for smell detection (Wu et al., 2024), we are not aware of their use in the context of JavaScript. Combined with the lack of modern benchmark datasets and standardized evaluation practices, this scenario complicates both empirical validation and tool evolution (Sharma et al., 2021).

To fill these gaps in the literature, this paper conducts a Systematic Literature Review (SLR) (Kitchenham and Charters, 2007) that not only catalogs existing tools, but also iden-

tifies blind spots and fosters convergence in detection strategies and evaluation methods tailored to the unique characteristics of JavaScript. Using well-known guidelines (Wohlin, 2014), we performed a structured search in four major databases (ACM Digital Library, IEEE Xplore, Scopus, and Springer) with a predefined search string. To ensure relevance, we applied strict filtering criteria, reducing the initial 1002 papers to 18 primary studies through duplicate removal, metadata filtering, and full-text review. Furthermore, we performed both backward and forward snowballing to expand the dataset, identifying 456 additional initial studies. After further filtering, we included 9 more primary studies, bringing the final dataset to 27 papers for in-depth analysis.

Leading software engineering conferences and journals, including ICSE (International Conference on Software Engineering), MSR (Mining Software Repositories), and TSE (Transactions on Software Engineering), have published studies on JavaScript code smell detection tools, highlighting a relevant academic interest in the topic. In particular, researchers published 22 papers on this topic from 2017 to 2023, with publication activity peaking at six papers in 2022. This SLR identified 22 code smell detection tools discussed in research papers. Through a detailed review of each tool, we collected key information, such as detected code smells, detection techniques, tool guides, implementation languages, and release years of the tools. Among the tools, researchers most frequently cited ESLint, JSLint, and JSHint, reflecting their widespread adoption and importance in the field.

Rule-based linting is the most common detection method since 55% of the tools use this approach. It offers efficiency for simple issues, but often misses more complex design problems (Gong et al., 2015). In particular, none of the reviewed tools employs AI-based techniques, although these approaches are widely used in other programming languages (Vo et al., 2025; Wu et al., 2024). Most tools are based on a Command-Line Interface (82%), while only 18% offer a Graphical User Interface. Regarding cost, 95% of the tools are free to use, reflecting the preference of the development community for open-source options. Furthermore, there is a growing trend toward framework-specific tools, which reflects the evolving landscape of JavaScript development.

The findings of this study have significant implications for both practitioners and researchers. For practitioners, most tools focus on simpler code smells, missing deeper architectural and complex code smells, such as Feature Envy, which can have an impact on maintainability. Tool builders should consider combining static and dynamic analysis and advocating for more adaptable tools suited to modern JavaScript frameworks. For researchers, the lack of AI-driven approaches and modern benchmark datasets presents opportunities to improve automated JavaScript code smell detection.

The remainder of this paper is organized as follows. Section 2 provides background information to support understanding of the study. Inspired by previous work (Fernandes et al., 2016; Moreira et al., 2022), Section 3 presents the study protocol, outlines the study goal, research questions, and experimental procedures. Section 4 presents the results of the study. Section 5 discusses threats to the study validity. Section 6 summarizes some related work. Finally, Section 7 concludes the paper and suggests future work.

2 Background

This section provides background information as follows. Section 2.1 discusses JavaScript and the progressive adoption of object-oriented programming by development teams. Section 2.2 defines code smells and examines recurring challenges in their detection.

2.1 JavaScript and Object-Oriented Programming

JavaScript is currently one of the most popular programming languages, as confirmed by the TIOBE Index [1] and GitHub surveys [2]. Additionally, the same GitHub survey showed that TypeScript, a strongly typed extension of JavaScript released in 2012, has gained significant traction, ranking third and surpassing Java for the first time. For instance, Scarsbrook et al. (2023) highlights trends in feature usage, showing how TypeScript evolves in real-world projects and influences modern software development.

ECMAScript 2015 is a major update to the JavaScript specification released in 2015 that brought new features and a cleaner syntax to the language (Zakas, 2016). For instance, this ECMAScript added native object-oriented features, making it easier to build complex software systems, but also increasing the risk of code smells (Johannes et al., 2019). Before ECMAScript 2015, developers emulated object-oriented principles, but the introduction of classes improved software reuse (Gallaba et al., 2017) and encouraged legacy code updates. Despite these changes, JavaScript remains dynamic and weakly typed, with first-class functions and prototype-based inheritance (Fard and Mesbah, 2013). Prototypes can change at runtime, affecting all related objects. The flexible object model of the language allows creating, modifying, and deleting properties at runtime, which makes it powerful but also prone to maintainability issues (Tómasdóttir et al., 2017).

Programming in JavaScript can be challenging and error-prone, especially in older codebases. Silva et al. (2017) analyzed eight real-world legacy JavaScript systems and found that outdated object-oriented patterns, such as emulating classes with functions and prototypes, are still widely used. When attempting to update these systems to the modern ECMAScript 2015 class syntax, they observed that only part of the code could be automatically migrated using a set of migration rules. Specifically, Silva et al. (2017) found that six out of the eight systems (75%) contained sections that required manual adjustments or could not be updated due to limitations of the language. The main issues encountered included function hoisting, incorrect use of constructors, and advanced features not supported by ECMAScript 2015. These factors contribute to making JavaScript more difficult to write, read, and maintain, increasing the risk of bugs in large or legacy projects.

The singularities that JavaScript code provides make it challenging for Web developers to write maintainable

[1] https://www.tiobe.com/tiobe-index/

[2] https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages

code without in-depth knowledge of the language. Consequently, JavaScript Web applications often contain many code smells (Saboury et al., 2017). Since manual detection of code smells is time-consuming and error-prone, automated detection tools can reduce long-term development costs and improve code maintainability (Fard and Mesbah, 2013).

2.2 Code Smells and Detection Strategies

A code smell refers to any symptom in the source code of a program that could indicate a problem (Marinescu, 2004). Refactoring aims to improve code maintenance by modifying the internal structure of a program to eliminate code smells (Fowler, 1999). Therefore, there is a direct relationship between refactoring and code smells. Researchers have thoroughly discussed this topic in the literature (Azeem et al., 2019; Santos et al., 2018; Tsantalis et al., 2008) due to the potential of this type of problem to compromise maintenance effort, cost, and software quality.

Due to time constraints or community-related factors, developers often lack the time or motivation to control the complexity of systems and devise effective design solutions before implementing their changes (Tamburri et al., 2019). However, some code smells are difficult to recognize even when there is motivation to detect them (Lanza and Marinescu, 2007). Therefore, they present a significant threat to the maintainability and costs of a software system. In fact, previous research (Abbes et al., 2011) showed that code smells significantly impact the ability of developers to understand the source code and make affected classes more susceptible to changes and faults.

Code smells can be identified by performing manual or automated analyzes in the source code (Rasool and Arshad, 2015). Most of the automated tools described in the literature employ heuristic methods for code smell detection, processing metrics, and filtering results based on predefined thresholds (Fard and Mesbah, 2013; Jia and Dittmer, 2017; Johannes et al., 2019). Regardless of the tool of choice, developers design automated strategies to minimize the subjectivity of manual processes while reducing the time and cost of software development projects (Tómasdóttir et al., 2018). Each tool differs from others in its choice of metrics or in the use of more advanced detection methods, such as machine learning techniques (Azeem et al., 2019).

Di Nucci et al. (2018) analyze how well machine learning techniques detect code smells by replicating and extending previous studies. They note that earlier research, such as Arcelli Fontana et al. (2016), reported high precision for machine learning-based detection. However, they argue that these results might be due to specific characteristics of the dataset rather than to the actual strength of the machine learning models. To test this, they used a more realistic dataset with multiple types of code smells instead of just one. Their results show that machine learning performance drops significantly in this setting, with F-Measure scores decreasing by up to 90%. Their results suggest that machine learning-based code smell detection is heavily dependent on dataset composition and may not work well in real world cases.

Researchers have examined this problem for some time, incorporating the perspective of the industry on the relevance of code smells in software development (Arcelli Fontana et al., 2016; Di Nucci et al., 2018; Sharma et al., 2021). For instance, Hozano et al. (2018) examined the differences in evaluation that can occur when different developers analyze the same code smell and what makes them agree on the problem classification. Their result show very little agreement among developers about what is a code smell, largely due to each developer's experience and exposure to previous issues. This result reinforces the importance of using automated methods to reduce cases of disagreement about the definition of a code smell or whether a problem should be fixed.

Schumacher et al. (2010) compared how professional developers and metric-based algorithms detect code smells in an industrial environment. Despite developers finding the task easy, their classification agreement was low, mainly influenced by misplaced methods. Metric-based detection performed well, suggesting that using automated pre-selection can reduce manual inspection effort. Combining automatic detection with manual review enhances the overall confidence in classification.

3 Study Protocol

This section describes our study protocol as follows. Section 3.1 introduces our study goal and research questions. Section 3.2 introduces our search string and describes the filtering criteria. Finally, Section 3.3 explains the steps we have performed to extract data from primary studies.

3.1 Goal and Research Questions

We rely on the Goal-Question-Metric template (Basili and Rombach, 1988) to formally define our study goal as follows: *analyze* the state-of-the-art of code smell detection tools 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 tools based on their features and detection strategies; *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* peer-reviewed academic publications since 1999, the year that the term code smell became popular (Fowler, 1999).

To achieve our study goal, we opted for a SLR based on strict literature guidelines (Kitchenham and Charters, 2007; Wohlin, 2014). The SLR process involves three main stages: planning, conducting, and reporting. In the planning stage, we identify the need for the review, formulate research questions, and establish a review protocol. In the conducting stage, we execute this protocol by selecting papers, extracting, and analyzing the data. In the final stage, we report our findings to the intended audience. In addition, we defined the Research Questions (RQs) below in order to guide our study.

RQ1: *What is the publication landscape of code smell detection tools in JavaScript since the popularization of the term Code Smell?* – Our first research question aims to understand the state of the art in the research community on the de-

velopment and evaluation of such tools. To address this gap, we analyze metrics including the number of publications over time, publication venues (e.g., conferences or journals), and the timeline of tool releases. These metrics provide a comprehensive view of the maturity, evolution, and research trends of the field.

RQ2: *Which code smell detection tools have been published so far and what are their main features?* – This research question presents the capabilities and limitations of existing JavaScript code smell detection tools, including detection strategies (e.g., rule-based, AI-driven), types of code smells detected, interface modalities (e.g., Command-Line or Graphical User Interface), and the availability of tools. These metrics enable a comparative analysis of tools and support the identification of both existing limitations and opportunities for future research and tooling improvements.

3.2 Search String and Filtering Criteria

We designed the search string to maximize the discovery of tools by considering variations of both the *code smell* and *JavaScript* terms. We not only included alternate keywords, such as bad smell and ECMAScript, but also employed the * wildcard to capture different word forms and suffixes. This approach ensured a comprehensive and flexible search string.

((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 developed our search string through six rounds of refinements. The initial version included specific tools and detection-related terms, which yielded only a few results. The number of results increased as we gradually removed these restrictive terms and focused on related code smell and JavaScript keywords. We then experimented various terms, such as "code clone" and "duplicated code", which raised the number of results but also shifted the focus away from our topic resulting in too many false positives. By incorporating terms related to test smells and including all relevant JavaScript synonyms, we assembled a balanced, final search string that yielded 1002 results. This iterative process ensured that our search was comprehensive without including excessive unrelated studies.

We rely on this search string to query primary studies in four electronic data sources: Scopus [3], ACM Digital Library [4], IEEE Xplore [5], and Springer [6]. We ran our final search string on May 31, 2025. We limited the search to metadata fields, including titles, abstracts, and keywords, to make later refinements easier due to the large number of results collected after each execution of the search string. We then imported BibTeX and text files into the JabRef tool [7] to organize references, and when necessary, we converted text files to BibTeX format.

We applied the same search string across all databases, adapting its format as needed to match the search requirements of each database. For instance, we divided the search string into individual search terms to comply with the advanced search interface of IEEE Xplore. We also performed a preliminary filtering process whenever possible, applying filters during the search. For instance, we restricted publication years to exclude papers published before 1999 according to our exclusion criteria (Table 1). These adjustments ensured consistency and relevance in the studies we recovered from each source.

Our automatic search yielded a total of 1,002 studies, with 220 papers collected from the ACM Digital Library, 104 from IEEE Xplore, 515 from Scopus, and 163 from Springer. We found many studies outside our research goal and, therefore, we applied filtering criteria. In order to restrict the included papers, we defined five inclusion criteria and three exclusion criteria. Table 1 shows the filtering criteria, including both inclusion and exclusion criteria. For instance, as exclusion criteria, we excluded studies that are not a peer-reviewed paper, such as those available in public repositories (e.g., arXiv). As an example of inclusion criteria, we only included studies that used or proposed a detection tool. We considered any software labeled as a tool by its authors.

Table 1. Filtering Criteria

Inclusion Criteria	• It proposes or mentions a code smell detection tool, a linting tool and/or a tool that fixes defects • It is focused on JavaScript language • It was written in English • It was published in Computer Science • It was accepted by two different researchers after full-text read
Exclusion Criteria	• It is not a primary study • It was published between 1999 and 2024 • It is not a paper (conference, symposium or workshop), journal article or magazine article

3.3 Data Extraction Steps

Figure 1 presents the results of the data extraction process. We applied a four-step refinement process to determine the final set of primary studies. In Refinement 1, we removed duplicates and non-academic works, such as technical reports and conference calls, reducing the initial set from 1002 to 312 studies. We rejected a paper whenever we found it matched an exclusion criteria. In Refinement 2, we reviewed the metadata, also applying inclusion and exclusion criteria, which

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

narrowed the selection to 24 studies. Refinement 3 involved a full-text review of these papers, eliminating those that did not propose or use a code smell detection tool, leaving 18 studies. We relied solely on manual reading and did not use software tools during the paper review and filtering process.

Finally, we performed both forward and backward snowballing processes (Kitchenham and Charters, 2007; Wohlin, 2014) in the 18 initial primary studies using the same filtering criteria presented in Section 3.2, identifying 9 additional relevant studies. For backward snowballing, we collected the references used in 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 used Google Scholar to identify papers that cited our primary studies, again applied the inclusion and exclusion criteria, and performed a full-text read on the included papers.

We performed an asynchronous voting system during all refinement steps: a paper could only advance to the next refinement step if two different researchers gave it positive inclusion votes. If the two researchers disagreed, a third researcher made the final decision. All authors participated in the data extraction process. Through this process and the application of our inclusion and exclusion criteria, we selected 27 papers for data extraction.

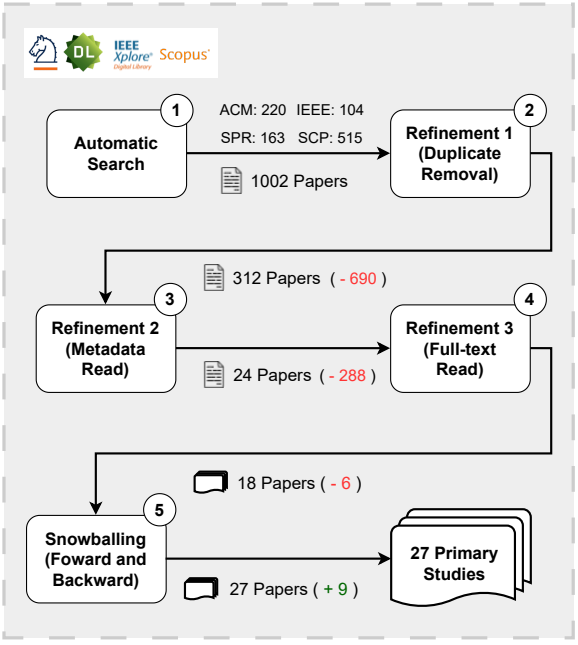

Figure 1. Number of Papers Filtered During Each Study Step

We prioritized papers focused on proposing detection tools; if unavailable, we relied on those using or citing such tools. To supplement our findings, we searched for additional information about the tools in online sources, including Google, GitHub, and other relevant repositories. When the release year of a tool was not available, we chose the publication year of the paper that introduced the tool. If neither was available, we selected the release year of the first commercial version. Lastly, if we could not find any of the previous data points, we used the earliest documented release year we found on the Web.

Table 2 presents the data extracted from the 22 tools collected by executing the SLR protocol (Kitchenham and Charters, 2007; Wohlin, 2014). We organized the extracted data for each tool into a spreadsheet to support the analysis, along with additional relevant information from the reviewed papers that could assist in the writing of this study. We encountered some challenges during data extraction, particularly with incomplete or missing tool descriptions. For instance, several papers did not clearly list the types of code smells detected or lacked a dedicated section outlining tool features such as supported languages or detection techniques. To address these gaps, we performed supplementary Web searches in an effort to gather as much information as possible about the 22 tools. Pairs of researchers performed the data extraction process and validated the collected information, maintaining double validation.

Table 2. Extracted Data From Each Tool and Its Description

Name	Description
Detected Code Smells	What code smells the tool detects?
Detection Technique	What is the problem modeling approach (Rule-based linting, Dynamic analysis, etc)?
Free for Use	Is the tool free for use?
Guide	Does the tool has a guide?
GUI	Does the tool has a Graphical User Interface?
Industry or Research	Does the tool was proposed by the industry or researchers?
Language Developed	In what language the tool was developed?
Number of Citations	Number of papers that cited the tool
Plug-in	Is the tool available as a plug-in?
Release Year	In which year was the tool released?
Still Maintained	Did the tool receive new updates in the last three years?
Tool Name	What is the tool name?

4 Results

In this section, we present the results of our study as follows. In Section 4.1, we describe the findings regarding the publishing landscape of code smell detection tools since the release of JavaScript. In Section 4.2, we present the results obtained after briefly studying the detection tools found. Finally, Section 4.3 describes the implications of the results for both practitioners and researchers.

4.1 Overview of Tools and Publications

Figure 2 shows the frequency of publications on the topic per year. In particular, researchers actively studied the topic from 2017 to 2023, with 22 publications during this time span

Table 3. Research Focus Areas and Paper Classification

Research Focus Area	Number of Papers	Papers List
Code Smell Detection	15	JSNOSE: Detecting JavaScript Code Smells (Fard and Mesbah, 2013) A Large-scale Empirical Study of Code Smells in JavaScript Projects (Johannes et al., 2019) An Empirical Study of Code Smells in JavaScript Projects (Saboury et al., 2017) On the Use of Smelly Examples to Detect Code Smells in JavaScript (Shoenberger et al., 2017) Detecting Code Smells in React-based Web Apps (Ferreira and Valente, 2023) Causal Inference of Server and Client-side Code Smells in Web Apps Evolution (Rio et al., 2024) Are Existing Code Smells Relevant in Web Games? an Empirical Study (Khanve, 2019) Automated Refactoring of Legacy JavaScript Code to ES6 Modules (Paltoglou et al., 2021) Code Smell Detection Tool for JavaScript Programs (Almashfi and Lu, 2020) DLint: Dynamically Checking Bad Coding Practices in JavaScript (Gong et al., 2015) The Adoption of Javascript Linters in Practice: A Case Study on ESLint (Tómasdóttir et al., 2018) Analyzing Linter Usage and Warnings Through Mining Software Repositories: A Longitudinal Case Study of JavaScript Packages (Heričko and Šumak, 2022) Let Us Lint: A Tool for Code Formatting And Code Enhancing (Vayadande et al., 2023) JSGuide: A Tool to Improve JavaScript Algorithms Focusing on IoT Devices (Oliveira and Mattos, 2022) GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications (Guarnieri, 2010)
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 (Bogner and Merkel, 2022) How and Why We End Up With Complex Methods: a Multi-language Study (Lopes and Hora, 2022) Anomaly Detection in Dynamic Programming Languages Through Heuristics Based Type Inference (Jia and Dittmer, 2017) BUGSJS: A Benchmark of JavaScript Bugs (Gyimesi et al., 2019) Detecting Unknown Inconsistencies in Web Applications (Ocariza et al., 2017) Detecting Common Weakness Enumeration Through Training the Core Building Blocks of Similar Languages Based on the CodeBERT Model (Park and Kim, 2023) DrAsync: Identifying and Visualizing Anti-patterns in Asynchronous JavaScript (Turcotte et al., 2022) JSOptimizer: An Extensible Framework for JavaScript Program Optimization (Liu, 2019)
Mining Software Repositories	2	Mining Rule Violations in JavaScript Code Snippets (Campos et al., 2019) Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild (Park et al., 2016)
Test Smell Detection	2	Investigating Test Smells in Javascript Test Code (Jorge et al., 2021) Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection (Fulcini et al., 2022)

and a peak of six papers in 2022. Our results did not reveal any studies on the topic published between 1999 and 2009. The increase in the number of publications between 2017 and 2023 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 (Rio et al., 2024). Furthermore, modern frameworks, such as React, Angular, and Vue.js, introduced new coding patterns, increasing the need for research on code smells (Ferreira and Valente, 2023). Lastly, the release of ECMAScript 2015, which added native object-oriented features, could have interested the research community that is familiar with studying code smells for languages with these features (Fowler, 1999; Fernandes et al., 2016; Marinescu, 2004; Johannes et al., 2019).

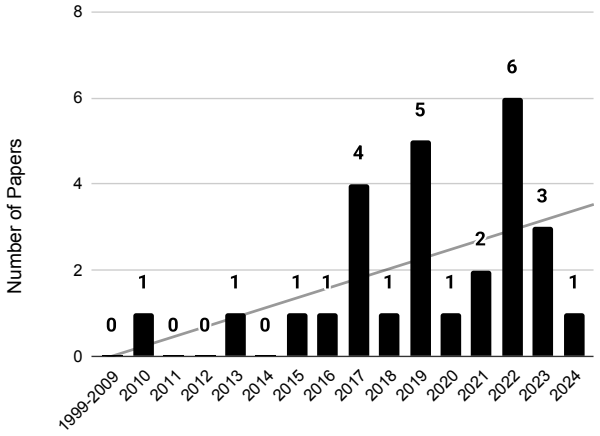

Figure 2. Frequency of Paper Publication by Year

Table 3 lists the 27 primary studies in five different research areas defined in this study by their main research topics: code smell detection (15 papers), JavaScript-specific code quality (8 papers), test smell detection (2 papers), and mining software repositories (2 papers). Researchers advanced the study of code smell detection from early static analysis (Fard and Mesbah, 2013) to framework-specific studies (Ferreira and Valente, 2023) and client-server smell detection (Rio et al., 2024). JavaScript-specific quality research explores TypeScript maintainability (Bogner and Merkel, 2022), async programming issues (Turcotte et al., 2022), and bug benchmarking (Gyimesi et al., 2019). Test smell detection remains under-explored but reveals issues such as asynchronous test failures (Jorge et al., 2021). Lastly, mining software repositories use GitHub data to assess JavaScript quality trends (Campos et al., 2019), confirming the value of empirical validation for practical development.

Table 4 and Figure 3 reinforce this result by illustrating the academic interest in code smell detection for JavaScript systems over the years. Early studies (2010–2017) focused on static analysis and heuristics, appearing in specialized venues, such as SANER (Saboury et al., 2017) and SCAM (Fard and Mesbah, 2013). Recent research (2017–2024) gained visibility in major conferences, such as ICSE (Turcotte et al., 2022), FSE (Khanve, 2019), and ASE (Ocariza et al., 2017), which emphasizes large-scale empirical validation. Journals, such as EMSE (Lopes and Hora, 2022; Rio et al., 2024) and TSE (Tómasdóttir et al., 2018), have published studies on the topic, including repository analyzes, expanding the field beyond niche discussions.

Although the number of studies on JavaScript code smell

Table 4. Primary Studies and Their Publishing Venues

Venue	Papers
Conference on Web Application Development (WebApps)	GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications (Guarnieri, 2010)
International Working Conference on Source Code Analysis and Manipulation (SCAM)	JSNOSE: Detecting JavaScript Code Smells (Fard and Mesbah, 2013)
International Symposium on Software Testing and Analysis (ISSTA)	DLint: Dynamically Checking Bad Coding Practices in JavaScript (Gong et al., 2015)
International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)	Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild (Park et al., 2016)
International Conference on Software Analysis, Evolution and Reengineering (SANER)	An Empirical Study of Code Smells in JavaScript Projects (Saboury et al., 2017)
Applications of Evolutionary Computation (EvoApplications)	On the Use of Smelly Examples to Detect Code Smells in JavaScript (Shoenberger et al., 2017)
Computing Conference (SAI)	Anomaly Detection in Dynamic Programming Languages Through Heuristics Based Type Inference (Jia and Dittmer, 2017)
International Conference on Automated Software Engineering (ASE)	Detecting Unknown Inconsistencies in Web Applications (Ocariza et al., 2017)
Transactions on Software Engineering (TSE)	The Adoption of Javascript Linters in Practice: A Case Study on ESLint (Tómasdóttir et al., 2018)
Software Quality Journal (SQJ)	A Large-scale Empirical Study of Code Smells in JavaScript Projects (Johannes et al., 2019)
European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)	Are Existing Code Smells Relevant in Web Games? an Empirical Study (Khanve, 2019)
International Conference on Software Testing, Validation and Verification (ICST)	BUGSJS: A Benchmark of JavaScript Bugs (Gyimesi et al., 2019)
International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)	JSOptimizer: An Extensible Framework for JavaScript Program Optimization (Liu, 2019)
International Conference on Mining Software Repositories (MSR)	Mining Rule Violations in JavaScript Code Snippets (Campos et al., 2019)
International Conference on Computer and Communication Systems (ICCCS)	Code Smell Detection Tool for JavaScript Programs (Almashfi and Lu, 2020)
Journal of Systems and Software (JSS)	Automated Refactoring of Legacy JavaScript Code to ES6 Modules (Paltoglou et al., 2021)
Symposium on Systematic and Automated Software Testing (SAST)	Investigating Test Smells in Javascript Test Code (Jorge et al., 2021)
Information and Software Technology (IST)	Detecting Code Smells in React-based Web Apps (Ferreira and Valente, 2023)
International Conference on Mining Software Repositories (MSR)	To type or not to type? A Systematic Comparison of the Software Quality of Javascript and Typescript Applications on Github (Bogner and Merkel, 2022)
Empirical Software Engineering (EMSE)	How and Why We End Up With Complex Methods: a Multi-language Study (Lopes and Hora, 2022)
International Convention on Information, Communication and Electronic Technology (MIPRO)	Analyzing Linter Usage and Warnings Through Mining Software Repositories: A Longitudinal Case Study of JavaScript Packages (Heričko and Šumak, 2022)
Symposium on Internet of Things (SIoT)	JSGuide: A Tool to Improve JavaScript Algorithms Focusing on IoT Devices (Oliveira and Mattos, 2022)
Automating Test Case Design, Selection and Evaluation (A-TEST)	Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection (Fulcini et al., 2022)
International Conference on Software Engineering (ICSE)	DrAsync: Identifying and Visualizing Anti-patterns in Asynchronous JavaScript (Turcotte et al., 2022)
Asia-Pacific Software Engineering Conference (APSEC)	Detecting Common Weakness Enumeration Through Training the Core Building Blocks of Similar Languages Based on the CodeBERT Model (Park and Kim, 2023)
International Conference on Computing Communication and Networking Technologies (ICCCNT)	Let Us Lint: A Tool for Code Formatting And Code Enhancing (Vayadande et al., 2023)
Empirical Software Engineering (EMSE)	Causal Inference of Server and Client-side Code Smells in Web Apps Evolution (Rio et al., 2024)

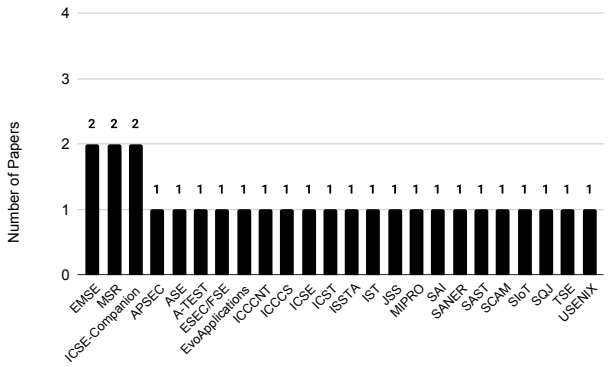

Figure 3. Frequency of the Topic Discussion in Conferences and Journals

detection is relevant, important gaps still exist. For instance, test smell detection remains underexplored, with only two papers addressing it (Fulcini et al., 2022; Jorge et al., 2021). Similarly, machine learning-based approaches, which are common in Java and Python (Arcelli Fontana et al., 2016; Thomas et al., 2013; Vatanapakorn et al., 2022), are still rare in JavaScript. This is surprising, given the presence of studies of AI techniques in other programming ecosystems (Vo et al., 2025; Wu et al., 2024). One reason for this gap may be the lack of recent datasets for JavaScript code smells, making it harder to evaluate models (Du et al., 2024). Another possible reason is the rapid evolution of the JavaScript ecosystem (Mitropoulos et al., 2019), including the proliferation of diverse frameworks and libraries (Bogner and Merkel, 2022), which can complicate the creation of generalized tools and approaches for smell detection (Barros and Adachi, 2021).

RQ1 Findings: Research on JavaScript code smell detection progressed from static analysis to framework-specific studies. There is a lack of studies on test smell detection, AI-based techniques, and refactoring tools focused on JavaScript systems. Top-tier venues (e.g., ICSE and FSE), specialized conferences (e.g., SANER and MSR), and journals (e.g., TSE and EMSE) published the 27 analyzed papers.

4.2 Feature of Detection Tools

Table 5 lists the 22 code smell detection tools identified in this systematic literature review in terms of their detection capabilities. Our results include tools focused on code smell detection for JavaScript systems, even if they provide additional features such as refactoring support. All tools rely on Abstract Syntax Trees for detection, so this information is not included in the Detection Technique column.

The results in Table 5 show the dominance of rule-based linting as the most popular detection technique, used by 55% of the tools. Popular tools, such as JSLint and ESLint, rely on predefined rules to analyze JavaScript systems (Tómasdóttir et al., 2017). Rule-based linting is efficient, making it

the preferred method for static analysis (Tómasdóttir et al., 2018). However, it has limitations, as it may miss more complex issues that require advanced pattern matching or runtime analysis (Gong et al., 2015). Although rule-based linting is the most common approach, some tools go beyond it by using static analysis tools to improve JavaScript maintainability checks. Tools, such as Google Closure Compiler, STEEL, and JSGuide, analyze code without running it, enabling detection of issues such as dead code, redundant operations, and improper JavaScript constructs (Fard and Mesbah, 2013; Jorge et al., 2021; Oliveira and Mattos, 2022).

However, dynamic analysis remains underused and only 5 out of 22 tools (23%) employ it. Tools such as DLint and DrAsync execute JavaScript in controlled environments to detect runtime issues that static analysis tools might overlook (Gong et al., 2015). 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 (Tang et al., 2023; Vo et al., 2025; Wang et al., 2020; Wu et al., 2024), we did not find a JavaScript code smell detection tool that uses machine learning or Large Language Models (LLMs). This gap suggests an opportunity to advance JavaScript analysis with modern AI-based approaches.

Regarding the code smells detected by the tools, Table 5 results shows that most do not detect complex, higher-level object-oriented smells, as described by Marinescu (2004). Instead, they primarily identify simpler issues, such as Long Method, Duplicated Code, and Dead Code. Tools, such as 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 tools analyzed, JSNose is one of the few capable of detecting Refused Bequest (Fard and Mesbah, 2013), a code smell related to improper inheritance in object-oriented programming (Fowler, 1999). This lack of complex smell detection suggests that most JavaScript analysis tools prioritize syntax and performance issues over maintainability and design complexity.

The Language Developed column in Table 5 indicates the programming language that the teams behind each tool chose for its implementation. Most of the teams chose JavaScript or Java, reflecting the prevalence of these languages and their alignment with language-specific analysis needs (Tómasdóttir et al., 2017). Other teams opted for Python, PHP, or C++, demonstrating that developers can adapt existing detection tools to the JavaScript programming environment depending on their project goals and expertise (Fernandes et al., 2016).

Table 6 presents relevant metadata information from the tools found in our primary studies. Three tools stood out as the most cited: ESLint (Johannes et al., 2019), JSLint (Paltoglou et al., 2018), and JSHint (Almashfi and Lu, 2020). The widespread adoption of JSLint and ESLint in industry, due to their versatility in code smell detection and unit testing, possibly contributed to their prominence in research (Tómasdóttir et al., 2018). JSNose also gained attention, possibly because Fard and Mesbah (2013) proposed one of the earliest approaches to code smell detection for JavaScript systems. Another notable point is the presence of general-purpose tools that detect smells in multiple programming languages, such as PMD and SonarQube (Fernandes et al., 2016; Shoenberger et al., 2017), which are widely used for Java. This result highlights their market relevance and adaptability to different languages.

The Guide column in Table 6 indicates whether the team behind each tool provided documentation or a user guide to assist users in installation, configuration, or usage. A publicly available guide can significantly improve the accessibility and adoption of a tool, especially for new users or those integrating it into their workflow for the first time (Aghajani et al., 2020). Among the 22 tools listed in Table 5, 13 provide user guides, demonstrating that more than half of the code smell detection tools analyzed offer some level of formal documentation for their users. However, the fact that nine tools lack documentation indicates that there is still significant room for improvement, particularly for research prototypes or niche tools.

The Free for Use column in Table 6 shows that, with the exception of Klocwork, all tools found in this study are free to use. This suggests that the JavaScript development community is heavily relying on open-source solutions to improve code quality (Fard and Mesbah, 2013). The availability of open-source tools also facilitates continuous improvements as new JavaScript features and frameworks emerge (Tómasdóttir et al., 2017). However, only 7 out of the 22 tools analyzed still receive updates. This suggests a general lack of long-term support for many tools in the ecosystem. This result presents an interesting pattern when looking at the two most cited tools, ESLint and JSLint, both of which are still maintained. Their active maintenance may partly explain their popularity, as developers tend to adopt tools that remain up-to-date with evolving language features and development practices (Heričko and Šumak, 2022).

Among the 22 tools presented in Table 6, we identified that researchers or academics developed exactly half (11 tools), while the software industry created the other half. 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 the tools that are still maintained come from the industry. This suggests that industry tools are more likely to be sustained over time, likely due to greater adoption, funding, and integration into real development environments (Tómasdóttir et al., 2017). Tools such as ESLint and SonarQube, which are still actively used and maintained, have strong community or company support, and are designed for daily use in production systems (Heričko and Šumak, 2022).

Lastly, 10 out of the 22 tools presented in Table 6 (45%) function as plug-ins, including JSHint, ESLint, and JSLint (Heričko and Šumak, 2022). These tools integrate into Integrated Development Environments, such as Visual Studio Code, providing real-time linting and code quality checks (Bogner and Merkel, 2022). The remaining 12 tools (55%) operate as standalone Command-Line Interface tools, primarily used in CI/CD pipelines rather than for direct developer feedback (Ghaleb et al., 2024). This distinction highlights a trade-off: plug-ins offer instant feedback to developers, while standalone tools enable deeper, batch-style analysis in automated workflows (Rostami Mazrae et al., 2023).

Table 7 shows the trends that can be observed from the release year of the tools. Older tools such as PMD and Sonar-

Table 5. Detection Capabilities of Code Smell Detection Tools

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

Table 6. Metadata of Code Smell Detection Tools

Tool Name	Number of Citations	Plug-in?	Free for Use?	Guide?	GUI	Release Year	Still Maintained?	Industry or Research?
JSLint	11	Yes	Yes	Yes	Yes	2013	Yes	Industry
ESLint	11	Yes	Yes	Yes	No	2013	Yes	Industry
JSHint	8	Yes	Yes	Yes	No	2011	Yes	Industry
JSNose	6	No	Yes	Yes	No	2013	No	Research
PMD	3	Yes	Yes	Yes	No	2002	Yes	Industry
JSOptimizer	2	No	Yes	No	No	2019	No	Research
JSCS	2	Yes	Yes	Yes	No	2013	No	Industry
WebScent	2	No	Yes	No	No	2012	No	Research
SonarQube	2	Yes	Yes	Yes	Yes	2007	Yes	Industry
FragilityLinter	1	Yes	Yes	No	Yes	2022	No	Research
JSGuide	1	No	Yes	No	No	2022	No	Research
STEEL	1	No	Yes	No	No	2021	No	Research
DrAsync	1	No	Yes	Yes	No	2021	No	Research
React Sniffer	1	No	Yes	Yes	No	2021	No	Research
Holocron	1	No	Yes	No	No	2017	No	Research
TypeDevil	1	No	Yes	No	No	2015	No	Research
Flow	1	Yes	Yes	Yes	No	2015	Yes	Industry
Closure Linter	1	No	Yes	Yes	No	2015	No	Industry
DLint	1	No	Yes	Yes	No	2015	No	Research
Google Closure Compiler	1	No	Yes	Yes	No	2009	No	Industry
TAJS	1	No	Yes	Yes	No	2009	No	Research
Klocwork	1	Yes	No	Yes	Yes	2001	Yes	Industry

Qube mainly detect simpler JavaScript code smells, such as Duplicated Code and Long Method. In contrast, newer tools such as React Sniffer and JSGuide analyze code based on modern development practices (Fulcini et al., 2022). For instance, React Sniffer detects React-specific smells, such as Large Component and Too Many Props, emphasizing the need for framework-aware static analysis (Ferreira and Valente, 2023). Similarly, JSGuide identifies the misuse of ECMAScript 2015 features, such as Rest Parameter Misuse and Spread Operator Misuse, highlighting a shift toward analyzing modern JavaScript features (Oliveira and Mattos, 2022).

RQ2 Findings: The analysis of 22 JavaScript code smell detection tools shows that rule-based linting dominates focusing on surface-level issues, such as Long Method and Dead Code. Recent tools, such as React Sniffer and JSGuide, target at modern JavaScript frameworks and features. Our results highlight gaps in the detection of complex smells, such as Shotgun Surgery, and the use of AI-driven techniques for code smell detection, presenting an opportunity for future studies.

Table 7. Evolution of Code Smell Detection Tools 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)
2016-2024	Focus on framework-specific smells, ECMAScript 2015 or newer features, and test smells (STEEL, DrAsync, React Sniffer, FragilityLinter, JSGuide)

4.3 Practical and Research Implications

The findings of this study have practical implications for both software practitioners and the research community, each group facing different challenges.

Implications to Practitioners: Software practitioners, including developers, technical leads, and DevOps engineers, often face the challenge of selecting and integrating tools that improve code quality without disrupting productivity (Tómasdóttir et al., 2017). This study shows that most JavaScript

code smell detection tools focus on surface-level issues, such as long methods or dead code, and are generally linters (e.g. ESLint, JSLint) (Johannes et al., 2019). These tools are highly suitable for projects that use a CI/CD workflow, where real-time feedback and fast performance are priorities (Rostami Mazrae et al., 2023). The best choice depends on factors such as team size, workflow automation, project stage, and codebase complexity (Heričko and Šumak, 2022).

As JavaScript codebases grow in complexity, especially with React, Node.js, and full-stack applications, traditional tools may not capture higher-level, context-dependent, or cross-file smells (Ferreira and Valente, 2023). This is where AI-based approaches, particularly those using LLMs, offer new possibilities for practitioners (Amaral et al., 2025). Unlike rule-based tools, LLM-powered detectors can potentially understand the intent behind code (Nam et al., 2024) and suggest refactorings that are more in line with human judgment (AlOmar et al., 2024; Amaral et al., 2025). This opens the door to more intelligent tooling that goes beyond syntax checks and style enforcement.

JavaScript developers could benefit from AI-assisted code review and refactoring suggestions, as developers of other languages already benefit (Amaral et al., 2025; Fan et al., 2025; Han et al., 2022; Lu et al., 2023). Large Language Models integrated into editors or CI/CD pipelines (e.g., via GitHub Copilot or custom GPT-based tools) (Nunes et al., 2025) could flag code smells, explain their impact on maintainability, and propose actionable refactorings, including context-aware transformations that rule-based linters might miss (Pornprasit and Tantithamthavorn, 2024).

Implications to Researchers: For researchers, this study presents a number of unresolved challenges and emerging research opportunities in the topic of JavaScript code smell detection. One of the most critical gaps is the absence of recent benchmark datasets for JavaScript, which can limit the replicability of experiments and the comparability of tools. This deficiency becomes especially problematic in the context of AI-driven techniques, whose training and validation depend on high-quality labeled data (Hou et al., 2024).

Although AI-based techniques, including machine learning, deep learning, and more recently LLMs, have advanced in other ecosystems (Vo et al., 2025; Wu et al., 2024), their adoption in JavaScript remains limited. This gap may exist due to the dynamic and weakly typed nature of the language (Wei and Ryder, 2013) and the rapid evolution of its ecosystem (Andreasen et al., 2017). However, these challenges present opportunities. AI can detect latent design flaws and uncover different types of code smells beyond the reach of rule-based tool.

Beyond AI, the field also lacks evaluation frameworks and user-centered studies. Few studies evaluate tools from the perspective of developer trust, adoption of refactoring, or perceived usefulness (Tómasdóttir et al., 2018). These aspects are critical to ensure that tools not only detect smells, but are also accepted and used effectively in real settings (Wohlin et al., 2012). Future research should invest in combining automated metrics with qualitative feedback from developers using tools in real-world projects.

Lastly, the fragmented nature of existing smell definitions with multiple taxonomies scattered across papers (Barros and Adachi, 2021), calls for a unified and evolving classification scheme, one that explicitly accounts for framework-specific and extensions of language code contexts (Rastogi et al., 2015). Doing so would not only improve tool precision but also foster collaboration and shared understanding in the research community.

5 Threats to Validity

We rely on strict guidelines (Wohlin et al., 2012) to discuss the threats to the study validity.

Construct Validity: For the SLR, 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 our inclusion and exclusion criteria for primary studies. To avoid deviating from the context of our study, we excluded incomplete tools or those that used only heuristic methods, which may have caused us to omit some tools.

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 contributed to reducing the likelihood of errors and ensured greater reliability in data extraction and organization.

External Validity: We decided to focus our work on tools specifically designed for software systems built using JavaScript rather than the so-called language-agnostic tools. This decision may have significantly limited our results, reducing the number of tools identified. To mitigate this threat, we used the guidelines of a SLR (Kitchenham and Charters, 2007), which describe that industrial tools not published in papers may be removed from the list of study objects.

6 Related Work

To our knowledge, no prior work has conducted a systematic literature review and a thorough evaluation of code smell detection tools for JavaScript software systems. However, we identified some related studies (Amaral et al., 2025; Barros and Adachi, 2021; Fernandes et al., 2016; Moreira et al., 2022; Pereira dos Reis et al., 2022).

Barros and Adachi (2021) investigated the definition and evolution of code smells in JavaScript. Analyzing the literature from 2013 to 2020, the authors identified 26 distinct code smells in eight publications. Despite JavaScript's growing prominence in areas like Web development, games, and 3D rendering, the study suggests a need for more empirical research to understand the impact of these code smells on JavaScript-based systems.

Pereira dos Reis et al. (2022) conducted a comprehensive analysis of existing approaches for identifying and visualiz-

ing code smells. The study reveals that the most frequent detection approaches are search-based (30.1%), metric-based (24.1%), and symptom-based (19.3%). Furthermore, it highlights that a significant majority of research (83.1%) uses open-source software, with Java being the primary language in 77.1% of the studies.

Amaral et al. (2025) investigated the application of LLMs for automated test smell refactoring in JavaScript. The study conducted an empirical evaluation of two LLM-based tools, GitHub Copilot Chat and Amazon CodeWhisperer, refactoring 148 test smell instances across ten real-world JavaScript projects. Their results showed that Copilot removed 58.78% of test smells and Whisperer 47.30%, while both tools generally preserved test behavior and code coverage. However, the authors observed that both systems also occasionally introduced new smells, such as "Lazy Test", indicating limitations in the current LLM capabilities.

Fernandes et al. (2016) presented a SLR and a comparative study of code smell detection tools. The authors identified 84 tools covering 61 different code smells in various programming languages and detection techniques, highlighting that most of the tools target Java, C, and C++. In addition, they conducted an in-depth comparison of four popular tools, finding that while detection results for the same code smell often overlapped, precision and recall varied significantly between tools and types of smells. The study also identified common usability challenges and provided guidelines to help developers and users select and improve code smell detection tools.

7 Conclusion

Code smells indicate potential software issues (Fowler, 1999). Detecting code smells in JavaScript systems is crucial as it directly impacts software maintainability and quality (Lanza and Marinescu, 2007). For instance, a duplicated code block in a Web application can lead to inconsistent updates and increased technical debt (Saboury et al., 2017). Although manual detection is possible, automatic tools with various strategies can assist in this task (Fard and Mesbah, 2013). This paper details a SLR on code smell detection tools (Sections 2 and 3) designed for JavaScript systems. We identified a total of 22 tools. This study provided a comprehensive review of JavaScript-specific code smell detection tools, emphasizing their characteristics and practical applications. As the first review of its kind, it bridges the gap between academic research and practical needs in software development. The findings may not only guide developers in selecting suitable tools, but also identify opportunities for innovation in tool development.

For future work in JavaScript code smell detection, we suggest a focus on advancing detection techniques and improving tool effectiveness. AI-driven approaches, including machine learning (Gesi et al., 2023) and LLMs (Amorim et al., 2025), could improve detection accuracy, especially for higher-level architectural smells. The lack of modern benchmark datasets remains a challenge, and creating a standardized dataset for evaluating tools would improve reproducibility. Furthermore, hybrid detection methods that combine static and dynamic analysis could provide more comprehensive results. As modern JavaScript frameworks continue to evolve, research should explore framework-specific smells (e.g., React, Vue.js) and their impact on maintainability. Finally, empirical validation using real-world repositories would bridge the gap between research and industry, ensuring that tools are practical to developers.

Declarations

Acknowledgments

This research was partially supported by Brazilian funding agencies: CNPq (Grant 406089/2025-6), CAPES, and FAPEMIG (Grant APQ-01488-24).

Authors' Contributions

All authors worked on writing and reviewing the paper, in addition to creating graphs, charts, and tables to facilitate understanding of the scientific process.

Availability of data and materials

The dataset generated and analyzed during the current study is available at `https://doi.org/10.5281/zenodo.15757269`

Competing interests

The authors declare that they have no competing interests.

References

Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15Th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE, IEEE Computer Society.

Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., and Shepherd, D. C. (2020). Software documentation: the practitioners' perspective. In *International Conference on Software Engineering (ICSE)*, pages 590–601.

Almashfi, N. and Lu, L. (2020). Code smell detection tool for java script programs. In *2020 5th International Conference on Computer and Communication Systems (ICCCS)*, pages 172–176. IEEE.

AlOmar, E. A., Venkatakrishnan, A., Mkaouer, M. W., Newman, C., and Ouni, A. (2024). How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*, pages 202–206.

Amaral, G., Gomes, H., Figueiredo, E., Bezerra, C., and Rocha, L. (2025). Improving javascript test quality with

large language models: Lessons from test smell refactoring. In *Brazilian Symposium on Software Engineering (SBES)*, pages 776–782. SBC.

Amorim, L., Costa, I., Alves, L., and Figueiredo, E. (2025). Bad smell detection using google gemini. In *Proceedings of the IEEE International Workshop on Advances in Artificial Intelligence and Machine Learning (AIML)*.

Andreasen, E., Gong, L., Møller, A., Pradel, M., Selakovic, M., Sen, K., and Staicu, C.-A. (2017). A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys (CSUR)*, 50(5):1–36.

Arcelli Fontana, F., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering (EMSE)*, 21:1143–1191.

Azeem, M. I., Palomba, F., Shi, L., and Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology (IST)*, 108:115–138.

Barros, A. and Adachi, E. (2021). Bad smells in JavaScript: A mapping study. In *Proceedings of the 9th Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pages 1–5. SBC.

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

Bogner, J. and Merkel, M. (2022). 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.

Campos, U. F., Smethurst, G., Moraes, J. P., Bonifácio, R., and Pinto, G. (2019). Mining rule violations in javascript code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 195–199. IEEE.

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., and De Lucia, A. (2018). Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621. IEEE.

Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., and Lou, Y. (2024). Evaluating large language models in class-level code generation. In *International Conference on Software Engineering (ICSE)*, pages 1–13.

Fan, L., Liu, J., Liu, Z., Lo, D., Xia, X., and Li, S. (2025). Exploring the capabilities of llms for code-change-related tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 34(6):1–36.

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). 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 18:1–18:12. Association for Computing Machinery.

Ferreira, F. and Valente, M. T. (2023). Detecting code smells in react-based web apps. *Information and Software Technology (IST)*, 155:107111.

Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2011). Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc.

Fulcini, T., Garaccione, G., Coppola, R., Ardito, L., and Torchiano, M. (2022). 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.

Gallaba, K., Hanam, Q., Mesbah, A., and Beschastnikh, I. (2017). Refactoring asynchrony in javascript. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 353–363.

Gesi, J., Shen, X., Geng, Y., Chen, Q., and Ahmed, I. (2023). Leveraging feature bias for scalable misprediction explanation of machine learning models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1559–1570. IEEE.

Ghaleb, T., Abduljalil, O., and Hassan, S. (2024). Ci/cd configuration practices in open-source android apps: An empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

Gong, L., Pradel, M., Sridharan, M., and Sen, K. (2015). Dlint: Dynamically checking bad coding practices in javascript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*.

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

Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszédes, A., Ferenc, R., and Mesbah, A. (2019). Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101. IEEE.

Han, X., Tahir, A., Liang, P., Counsell, S., Blincoe, K., Li, B., and Luo, Y. (2022). Code smells detection via modern code review: A study of the openstack and qt communities. *Empirical Software Engineering*, 27(6):127.

Heričko, T. and Šumak, B. (2022). 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. IEEE.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. (2024). Large language models for software engineering: A systematic literature review. *Transactions on Software Engineering and Methodology (TOSEM)*, 33(8):1–79.

Hozano, M., Garcia, A., Fonseca, B., and Costa, E. (2018). Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology (IST)*, 93:130–146.

Jensen, S. H., Madsen, M., and Møller, A. (2011). Modeling the html dom and browser api in static analysis of javascript web applications. In *19th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, page 59–69.

Jia, X. and Dittmer, H. (2017). Anomaly detection in dynamic programming languages through heuristics based type inference. In *2017 Computing Conference*, pages 286–293. IEEE.

Johannes, D., Khomh, F., and Antoniol, G. (2019). A large-scale empirical study of code smells in javascript projects. *Software Quality Journal (SQJ)*, 27:1271–1314.

Jorge, D., Machado, P., and Andrade, W. (2021). Investigating test smells in javascript test code. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing (SAST)*, pages 36–45.

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

Kitchenham, B. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University and University of Durham.

Lanza, M. and Marinescu, R. (2007). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

Liu, Y. (2019). 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. IEEE.

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

Lu, J., Yu, L., Li, X., Yang, L., and Zuo, C. (2023). Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 647–658. IEEE.

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

Mitropoulos, D., Louridas, P., Salis, V., and Spinellis, D. (2019). Time present and time past: analyzing the evolution of javascript code in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 126–137. IEEE.

Moreira, R., Fernandes, E., and Figueiredo, E. (2022). Review-based comparison of design pattern detection tools. In *Proceedings of the 29th International Conference on Pattern Languages of Programs (PLoP)*, pages 17:1–17:16. The Hillside Group.

Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., and Myers, B. (2024). Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13.

Nunes, H., Figueiredo, E., Rocha, L., Nadi, S., Ferreira, F., and Esteves, G. (2025). Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 669–680. IEEE.

Ocariza, F. S., Pattabiraman, K., and Mesbah, A. (2017). Detecting unknown inconsistencies in web applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 566–577. IEEE.

Oliveira, F. L. and Mattos, J. C. (2022). Jsguide: A tool to improve javascript algorithms focusing on iot devices. In *2022 Symposium on Internet of Things (SIoT)*, pages 1–4. IEEE.

Paltoglou, A., Zafeiris, V. E., Giakoumakis, E. A., and Diamantidis, N. (2018). 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. IEEE.

Paltoglou, K., Zafeiris, V. E., Diamantidis, N., and Giakoumakis, E. A. (2021). Automated refactoring of legacy javascript code to es6 modules. *Journal of Systems and Software (JSS)*, 181:111049.

Park, C. and Kim, R. Y. C. (2023). 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. IEEE.

Park, J., Lim, I., and Ryu, S. (2016). 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.

Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G., and Anslow, C. (2022). Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering*, 29(1):47–94.

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

Rasool, G. and Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895.

Rastogi, A., Swamy, N., Fournet, C., Bierman, G., and Vekris, P. (2015). Safe & efficient gradual typing for typescript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 167–180.

Rio, A., Abreu, F. B. e., and Mendes, D. (2024). Causal inference of server-and client-side code smells in web apps evolution. *Empirical Software Engineering (EMSE)*, 29(5):133.

Rostami Mazrae, P., Mens, T., Golzadeh, M., and Decan, A. (2023). On the usage, co-usage and migration of ci/cd tools: A qualitative analysis. *Empirical Software Engineering (EMSE)*, 28(2):52.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. (2017). 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. IEEE.

Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., Do Nascimento, R. S., Freitas, M. F., and De Mendonça, M. G. (2018). A systematic review on the code smell effect. *Journal of Systems and Software (JSS)*, 144:450–477.

Scarsbrook, J. D., Utting, M., and Ko, R. K. (2023). Typescript's evolution: An analysis of feature adoption over time. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 109–114. IEEE.

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., and Shaw, M. (2010). Building empirical support for automated code smell detection. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10.

Sharma, T., Efstathiou, V., Louridas, P., and Spinellis, D. (2021). Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software (JSS)*, 176:110936.

Shoenberger, I., Mkaouer, M. W., and Kessentini, M. (2017). On the use of smelly examples to detect code smells in javascript. In *Applications of Evolutionary Computation: 20th European Conference, EvoApplications*, pages 20–34.

Silva, L. H., Valente, M. T., and Bergel, A. (2017). Refactoring legacy javascript code to use classes: The good, the bad and the ugly. In *International Conference on Software Reuse (ICSR)*, pages 155–171. Springer.

Tamburri, D. A., Palomba, F., Serebrenik, A., and Zaidman, A. (2019). Discovering community patterns in open-source: a systematic approach and its evaluation. *Empirical Software Engineering (EMSE)*, 24:1369–1417.

Tang, W., Tang, M., Ban, M., Zhao, Z., and Feng, M. (2023). Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software (JSS)*, 199:111623.

Thomas, S. W., Nagappan, M., Blostein, D., and Hassan, A. E. (2013). The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering (TSE)*, 39(10):1427–1443.

Tómasdóttir, K. F., Aniche, M., and Van Deursen, A. (2017). Why and how javascript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 578–589. IEEE.

Tómasdóttir, K. F., Aniche, M., and Van Deursen, A. (2018). The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering (TSE)*, 46(8):863–891.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *2008 12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–331. IEEE.

Turcotte, A., Shah, M. D., Aldrich, M. W., and Tip, F. (2022). Drasync: identifying and visualizing anti-patterns in asynchronous javascript. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pages 774–785.

Van Emden, E. and Moonen, L. (2002). Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE.

Vatanapakorn, N., Soomlek, C., and Seresangtakul, P. (2022). Python code smell detection using machine learning. In *2022 26th International Computer Science and Engineering Conference (ICSEC)*, pages 128–133. IEEE.

Vayadande, K., Mukhopadhyay, K., Chaudhari, V., Manwadkar, S., Mutalik, T., and Gawali, I. (2023). 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. IEEE.

Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., and Oizumi, W. (2015). 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. IEEE.

Vo, Q.-H., Dao, H., and Fukuda, K. (2025). Harnessing the power of llms for code smell detection in terraform infrastructure as code. In *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 533–542. IEEE.

Wang, W., Li, G., Ma, B., Xia, X., and Jin, Z. (2020). 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. IEEE.

Wei, S. and Ryder, B. G. (2013). Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, page 336–346, New York, NY, USA. Association for Computing Machinery.

Wohlin, C. (2014). 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.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering (ESE)*. Springer Science & Business Media, 1st edition.

Wu, D., Mu, F., Shi, L., Guo, Z., Liu, K., Zhuang, W., Zhong, Y., and Zhang, L. (2024). 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.

Zakas, N. C. (2016). *Understanding ECMAScript 6: the definitive guide for JavaScript developers*. No Starch Press.