# Déjà Vu: A Replication Study on Code Smells and Faults in JavaScript Projects

Kevin Pacifico
k.pacifico@studenti.unisa.it
University of Salerno
Fisciano, Italy

Giammaria Giordano
giammaria.giordano@pegaso.it
Pegaso University
Naples, Italy
University of Salerno
Fisciano, Italy

Valeria Pontillo
valeria.pontillo@gssi.it
Gran Sasso Science Institute
L'Aquila, Italy

Massimiliano Di Penta
dipenta@unisannio.it
University of Sannio
Benevento, Italy

Damian A. Tamburri
datamburri@unisannio.it
University of Sannio
Benevento, Italy

Fabio Palomba
fpalomba@unisa.it
University of Salerno
Fisciano, Italy

## ABSTRACT

Code smells are symptoms of poor design choices that can harm software quality. Their relation to fault-proneness has been studied in statically typed languages, such as Java, but less so in dynamic languages like JavaScript, which are becoming increasingly central given their primary role in rendering AI models accessible to their intended audience. Previous work on JavaScript was limited in scope, which affected the generalizability of its findings. This paper replicates and extends Johannes *et al.*'s study *"A Large-scale Empirical Study of Code Smells in JavaScript Projects"* to examine how code smells impact the fault-proneness of JavaScript applications. We analyze a large sample of 50 projects and nearly 100k commits across multiple domains, applying survival analysis with Cox models and robustness checks. We confirm that files with smells, such as *Variable Re-Assignment*, *Complex Code*, and *Conditional Assignment*, are more prone to faults. We also find that smell survivability varies across projects and that smells introduced at file creation often persist. These results offer a more ecologically valid and replicable perspective on the impact of code smells on JavaScript systems.

## CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

## KEYWORDS

Code smells; Fault-proneness; JavaScript; Software Maintenance and Evolution.

**ACM Reference Format:**
Kevin Pacifico, Giammaria Giordano, Valeria Pontillo, Massimiliano Di Penta, Damian A. Tamburri, and Fabio Palomba. 2026. Déjà Vu: A Replication Study on Code Smells and Faults in JavaScript Projects. In . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

JavaScript has long been the most widely used programming language worldwide [33], supporting an ecosystem that spans web development to mobile and desktop applications. Its centrality is further reinforced today by the diffusion of large language models (LLMs), for which JavaScript is among the most frequently generated outputs.[1] In practice, this means that JavaScript is not only the language through which most modern systems are rendered accessible to end users, but also one whose quality directly affects the reliability of AI-assisted development. This makes it particularly urgent to understand how design flaws emerge and persist in JavaScript systems, especially given the prevalence of code smells—design symptoms that can indicate deeper structural problems and are associated with higher fault-proneness [12].

Beyond their impact on maintainability, such flaws also affect code comprehension, a cornerstone of both human reasoning and automated analysis. Poorly structured or inconsistent code hinders developers' ability to understand system behavior and intentions, slowing maintenance and evolution. At the same time, it can hamper the performance of LLM-based tools, which depend on code readability and consistency to produce accurate summaries, repairs, and recommendations.

Despite the practical relevance of the problem, the software engineering (SE) research community has primarily focused on statically typed languages such as Java [3, 8, 16, 40], leaving dynamic languages like JavaScript comparatively less studied. This imbalance is striking, as it overlooks a language that is both ubiquitous in practice and increasingly central, especially in the era of AI-assisted programming.

A notable exception is the study by Johannes *et al.* [18], which represents one of the first systematic efforts to empirically examine code smells in JavaScript systems. Their work provides initial evidence of how smells manifest in real-world projects and their relationship to software quality. Analyzing more than 1,800 releases from 15 popular open-source projects, the authors demonstrated

[1] https://medium.com/%40alinaqishaheen/top-programming-languages-for-ai-coding-assistance-ranked-9d69ff03e082

that files affected by code smells are considerably more prone to faults, with specific smells, such as *Variable Re-Assignment*, *Assignment in Conditional Statements*, and *Complex Code*, being strongly associated with faults. At the same time, their release-based perspective could not capture issues introduced and resolved between releases, and the limited project sample reduces ecological validity.

In this replication study, we extend their work by examining 97,636 commits across 50 real-world projects of varying sizes, domains, and contributor communities. Our commit-level perspective allows us to capture a more fine-grained evolution of code smells and faults, revealing phenomena that remain invisible in release-based designs. Our results confirm that smelly files are generally more fault-prone and provide new insights. Specifically, smell survivability varies significantly across projects, and smells introduced at file creation tend to persist, amplifying their long-term impact on code quality. By offering a broader and more detailed account of how code smells emerge and persist, this study contributes evidence that is increasingly pressing, given the language's ubiquity and its growing role in AI-assisted software development.

## 2 RELATED WORK

Fowler and Beck originally defined code smells as symptoms of poor design that may signal deeper structural problems in software systems [12]. Building on this definition, the software engineering community has investigated code smells from multiple perspectives, particularly in the context of Java.

Several studies explored the evolutionary dimension of code smells. Giordano *et al.* analyzed their relationship with design mechanisms such as design patterns, inheritance, and delegation, finding that certain smells are positively correlated with these techniques [14, 15]. Tufano *et al.* [35] examined when and why smells arise, showing that they are often introduced in the earliest stages of development and are typically removed only when files themselves are deleted. Other works investigated the consequences of smells: Li and Shatnawi [29] linked them to a higher likelihood of bugs, while Sjoberg *et al.* [31] found only limited effects on maintenance effort. In contrast, Abbes *et al.* [1] reported negative effects on code understandability. Khomh *et al.* [20] and Palomba *et al.* [25] provided further evidence that classes affected by code smells are more change- and fault-prone, though the benefits of removing smells are not always clear-cut.

While these studies provide substantial evidence for statically typed languages, far fewer works have addressed dynamic languages such as JavaScript. This is an important gap, since JavaScript has unique language features and is increasingly central in both traditional development and AI-assisted programming. Our study contributes to filling this gap by replicating and extending prior research in the context of JavaScript, offering new evidence on the prevalence, persistence, and impact of smells in real-world projects.

Research specifically on JavaScript code smells has largely focused on tool support. Fard *et al.* [10] introduced JNose, capable of detecting 13 smell types, with *Lazy Object* and *Long Method* emerging as the most common. Nguyen *et al.* [23] developed a tool to detect intermixing issues across HTML, CSS, and JavaScript, while static analyzers such as ESLint,[2] JSLint [5], and JSHint [4] enforce coding conventions and best practices.

Complementary to detection, Saboury *et al.* [28] conducted an empirical study on the impact of 12 code smells in JavaScript server-side applications. By analyzing 537 releases across five projects and surveying over 1,400 developers, they showed that smells such as *Variable Re-Assignment*, *Assignment in Conditional Statements*, and *Nested Callbacks* increase fault-proneness and hinder maintainability. Their findings highlight the practical relevance of smells in JavaScript and resonate with later evidence from Johannes *et al.* [18] and with our own replication study.

**Table 1: Code Smells Detectable by the Framework of Johannes *et al.* [18], grouped by category.**

| Code Smell | Description |
|---|---|
| **Syntactic Concerns** | |
| Lengthy Lines | A single line of code contains too many characters, reducing readability and increasing the likelihood of horizontal scrolling and visual clutter. |
| Chained Methods | Excessive method chaining within a single statement makes debugging harder and obscures the individual method effects, especially when side effects are involved. |
| Long Parameter List | Functions with many parameters are harder to read, test, and call correctly, often suggesting that the function does too much or lacks proper abstraction. |
| Long Method | Methods that span many lines are often harder to understand, maintain, and reuse, and tend to violate the single-responsibility principle. |
| Complex Switch Case | Switch statements with many cases become unwieldy and error-prone, often indicating the need for better abstraction or use of polymorphism. |
| **Structural Complexity** | |
| Nested Callbacks | Deeply nested asynchronous callbacks—also known as "callback hell"—make control flow hard to follow and error handling difficult. |
| Depth Smell | Excessive levels of indentation indicate deep nesting, which reduces readability and signals high structural complexity. |
| Complex Code | High cyclomatic complexity makes the code harder to test, understand, and maintain, increasing the likelihood of faults. |
| Extra Bind | Unnecessary use of `.bind(ctx)` introduces redundant code and may confuse readers about whether the context is actually needed. |
| This Assign | Assigning `this` to a variable such as `self` creates ambiguity in scope management and can signal outdated patterns, especially in modern JavaScript that uses arrow functions. |
| **Semantic Misuses** | |
| Variable Re-Assignment | Reassigning a variable with a value of a different type or purpose in the same scope undermines code clarity and can lead to type-related bugs in dynamically typed languages. |
| Assignment in Conditional Statements | Using an assignment instead of a comparison inside conditionals may result in unintended logic and hard-to-detect bugs, often due to typographical errors. |

## 3 THE REFERENCE WORK

The reference work for our replication is the empirical study by Johannes *et al.* [18], which represents a foundational effort in analyzing the role of code smells in the context of JavaScript development. Their study was among the first to examine the presence, impact, and evolution of code smells tailored to the idioms and characteristics of a language like JavaScript, offering a reusable analytical framework that has informed subsequent research.

The study addresses three research questions: (1) whether files affected by code smells are more fault-prone than clean files; (2) whether different types of code smells exhibit varying levels of fault-proneness; and (3) how long code smells tend to persist over time. To address these questions, the authors analyze 1,807 releases across 15 popular open-source JavaScript projects. They develop a static analysis framework based on ESLint to detect a catalog of

[2] https://eslint.org/

12 JavaScript-specific code smells, derived from widely adopted community style guides.

This smell catalog covers both stylistic and semantic concerns frequently encountered in JavaScript code. Table 1 provides an overview of the 12 smells and their corresponding descriptions that can be grouped into three broad categories: *Syntactic concerns*, referring to surface-level violations of readability and formatting conventions (e.g., excessively long lines, method chaining); *Structural complexity*, which includes smells that increase cognitive load through deep nesting or convoluted control flow (e.g., *Nested Callbacks*, *Depth Smell*); and *Semantic misuses*, capturing patterns that may lead to confusion or unintended behavior (e.g., *Variable Re-Assignment* with conflicting meanings or assignment operators in conditional expressions).

To assess the relationship between code smells and software quality, Johannes *et al.* identify fault-inducing changes using the SZZ algorithm [32], combined with heuristics proposed by Da Costa *et al.* [6] to reduce false positives in bug-fixing commit detection. The approach relies on commit message patterns to identify fixes, which are then traced back using `git blame` to associate faults with the lines that introduced them. The authors then apply survival analysis using the *Cox Proportional Hazards Model* [21] to estimate the relative risk of faults in smelly versus smell-free files. Their results indicate that smelly files are, on average, 33% more fault-prone than clean ones and 45% more fault-prone when considering their dependencies with other files. Certain smells, i.e., *Variable Re-Assignment*, *Assignment in Conditional Statements*, and *Complex Code*, stand out as particularly fault-prone. Furthermore, code smells are often introduced early in the lifecycle of a file and tend to persist across multiple releases, raising concerns about long-term maintainability.

**Replication Statement**

While the study by Johannes *et al.* [18] provides important empirical insights and a reusable methodology, our replication is motivated by the need to assess whether its findings hold across different development contexts. The original analysis focuses on 15 highly popular and mature projects, conducting survival analysis at the release level to ensure data quality, stability, and sufficiently long evolutionary histories. This design, while well-suited for an exploratory investigation, inherently captures system states at release points, meaning that issues introduced and resolved between two releases are not observable.

Replications play a crucial role in strengthening the evidence base around empirical claims by evaluating whether observed patterns remain consistent across broader and more diverse samples. In this spirit, our study adopts the same methodological framework while substantially expanding the dataset to 50 projects and 97,636 commits, and conducting the analysis at the commit level. This approach captures finer-grained evolution and encompasses a wider range of JavaScript ecosystems, development styles, and contributor communities.

## 4 RESEARCH METHOD

The *goal* of this study is to assess the ecological validity of the findings of Johannes *et al.* [18] on the relationship between code smells and fault-proneness in JavaScript systems, using a broader and more diverse set of open-source projects. The *quality focus* is on assessing whether previously observed correlations between specific code smells and the likelihood of faults hold across different development contexts and project characteristics. The *perspective* is of both researchers and practitioners: the former benefits from a replication that strengthens the evidence base and explores the generalizability of earlier results, while the latter gains a deeper understanding of how persistent code smells may affect maintainability and fault risk in a broader range of real-world systems.

Following the original study [18], our investigation centers on the same three research questions that aim to understand how code smells affect the fault-proneness of JavaScript applications.

> ***RQ$_1$. Comparison of Fault-Proneness Between Smelly and Non-Smelly Files.*** *Is the fault-proneness risk higher in files with code smells than for those without code smells?*

The goal of the **RQ**$_1$ is to compare the failure time between JavaScript files with and without code smells. The study conducted by Johannes *et al.* [18] on 15 projects revealed that files without code smells have a 33% lower risk of failure and 45% when dependencies with other files are considered. By expanding the analysis to 50 projects and changing the granularity from releases to commits, we aim to confirm or revise these percentages.

> ***RQ$_2$. On the varying fault-proneness of different smell types.*** *Are JavaScript files with code smells equally fault-prone?*

**RQ**$_2$ aims to identify which code smells have the greatest impact on software quality, helping to determine which should be prioritized during refactoring. The study conducted by Johannes *et al.* [18] found that the smells *Variable Re-Assignment*, *Assignment in Conditional Statements*, and *Complex Code* are among those most strongly associated with a high fault-proneness. By extending the analysis to 50 JavaScript projects, we aim to assess whether these findings hold across a broader dataset or if additional high fault-prone smell patterns emerge.

> ***RQ$_3$. On code smell survivability.*** *How long do code smells survive in JavaScript projects?*

**RQ**$_3$ explores the survivability of code smells by analyzing when they are introduced and how long they persist in JavaScript files. The study by Johannes *et al.* [18] revealed that many code smells are present since a file is created and tend to persist over time. Among them, *Variable Re-Assignment* emerged as the most long-lived code smell. By expanding the analysis to a larger number of projects and commits, we aim to assess whether the lifespan of code smells varies depending on the type or complexity of the project.

When conducting our empirical experiment, we adopted the Software Engineering practices described by Wohlin *et al.* [39]. In terms of reporting, we leverage the *ACM/SIGSOFT Empirical Standards*.[3] Specifically, we used the "General Standard", "Data Science", and "Repository Mining" guidelines.

Figure 1 overviews the research method of this study. We started by selecting open-source JavaScript repositories from GitHub. Then, we ran the framework of Johannes *et al.* [18] and extracted

[3] https://github.com/acmsigsoft/EmpiricalStandards

information about smelliness, including the introduction and removal of code smells, as well as information about faults. After data extraction, the framework applied the *Cox Proportional Hazards* model [13] to evaluate the correlation between code smells and fault proneness over time.

## 4.1 Project Selection

The selection of software systems was driven by various considerations. First, we used GitHub Search [7], a specialized platform designed to identify open-source projects hosted on GitHub. Given the focus of our study, we randomly selected 50 JavaScript projects, applying a minimum threshold of 800 stars as a filtering criterion. Our motivation for using stars as a filtering criterion is that they have been shown to be a reliable proxy for estimating both the popularity of repositories and their quality [27].

**Table 2: Statistical Description of Projects Analyzed.**

| Statistic | Issues | Stars | Contributors | LOC |
|---|---|---|---|---|
| Mean | 1,143.06 | 17,553.28 | 199.14 | 138,621.4 |
| Standard Deviation | 768.18 | 15,614.79 | 154.98 | 206,071.1 |
| Min | 2 | 864 | 39 | 3,041 |
| 25th Percentile | 648.25 | 10,425 | 90 | 29,808.25 |
| Median (50%) | 1,227 | 14,900 | 157.5 | 65,314 |
| 75th Percentile | 1,504.75 | 19,800 | 265.5 | 146,356 |
| Max | 4,752 | 106,000 | 836 | 1,252,611 |

Table 2 provides the statistical description of the projects analyzed. The median number of stars is 14,900, while the maximum value is 106,000, suggesting the presence of outliers. The number of contributors shows variability, with a median of 157.5 and a maximum of 836. The distribution of lines of code is extensive: although the median is 65,314, some projects exceed 1.2 million, while others remain below 10,000. These results highlight the heterogeneity of the dataset, which includes both lightweight and extremely large-scale projects.

Moreover, the dataset is heterogeneous in terms of its functional scope as well. The selected projects can be grouped into eight main categories: documentation and code-quality tools (8 projects), frameworks and development utilities (8 projects), build systems and automation (5 projects), data management and persistence (3 projects), validation libraries (4 projects), networking solutions (6 projects), user interface and visualization libraries (10 projects), and a miscellaneous group of specialized toolkits (6 projects).

By encompassing this broad functional spectrum, the dataset not only reflects the diversity of modern JavaScript development but also mitigates potential biases that could arise from focusing on a single application domain or project type. As a result, the ensuing analysis is grounded in a representative cross-section of real-world software artifacts, enabling more generalizable insights into widely-adopted development practices, maintenance patterns, and ecosystem-wide trends. Furthermore, this heterogeneity has direct implications for code comprehension. Projects that differ in purpose, size, and architectural complexity pose distinct cognitive challenges to developers, influencing how code is navigated, understood, and maintained over time.

## 4.2 Operationalization of the Research Method

In the following, we outline the research method employed to address the research question. To conduct our investigation, we employed the framework proposed by Johannes *et al.* [18] to extract information about faults, code smells, and survivability analysis through three dedicated modules.

***RQ*$_1$. *Comparison of Fault-Proneness Between Smelly and Non-Smelly Files.*** Building the experimental design of Johannes *et al.* [18], we addressed the first research question by performing survival analysis. Specifically, we compare the time until a fault occurs in files containing code smells with those that have a match between faulty and smelly lines, and files without code smells. For each file and for each commit $c$ (i.e., corresponding to a commit), we also compute the following metrics: (i) **Time** in terms of the number of hours between the previous commit of the file and the commit $c$; (ii) **Smelly** with a binary outcome, i.e., 1 for commit $c$ with a file containing a code smells, 0 otherwise; and (iii) **Event** that is equal to 1 if the commit $c$ is a fault-fixing change and if there is at least one match between the faulty lines and the code smell lines, 0 otherwise. If there is no match, we consider the fault-fixing change unrelated to code smells.

Using the smelly metric, we split the data into two groups: one containing files with code smells (smelly = 1) and another containing files without any of the 12 studied code smells (smelly = 0). For each group, we create an individual *Cox Proportional Hazard Model*, i.e., a statistical method used in survival analysis to examine how various factors influence the time until an event occurs.

***RQ*$_2$. *On the varying fault-proneness of different smell types.*** To evaluate whether all code smells contribute equally to fault-proneness, the framework adopts a research method similar to the one used in **RQ**$_1$, but in a more fine-grained manner. Rather than considering code smells as a single aggregated factor, the framework considers the effect of each code smell category separately, enabling the assessment of the specific impact each type of smell has on fault occurrence. Therefore, for each smell category, the framework defines a binary *Smelly* metric, which indicates whether the corresponding smell is present in a given file at a particular commit. This disaggregated representation allows the survival model to estimate the unique contribution of each smell type to the overall hazard of fault introduction. In addition to the presence or absence of specific smells, the model accounts for several control variables that could influence the likelihood of a fault. These include the lines of code (LOC), the extent of code churn [22], and the number of previously reported faults associated with the file, which serves as a proxy for its historical effectiveness.

For each subject system, the framework constructs a *Cox Proportional Hazards Model* and performs tests for non-proportionality to verify that the model's assumptions are satisfied and that the hazard ratios can be interpreted reliably. This approach enables the determination of whether certain code smells are more closely correlated with software faults than others, and whether some smells represent a disproportionately higher risk to software reliability.

***RQ*$_3$. *Code Smell Identification and Survivability.*** To address **RQ**$_3$, the framework tracks all commits that modify smelly files, enabling the monitoring of smell evolution over time. Specifically,

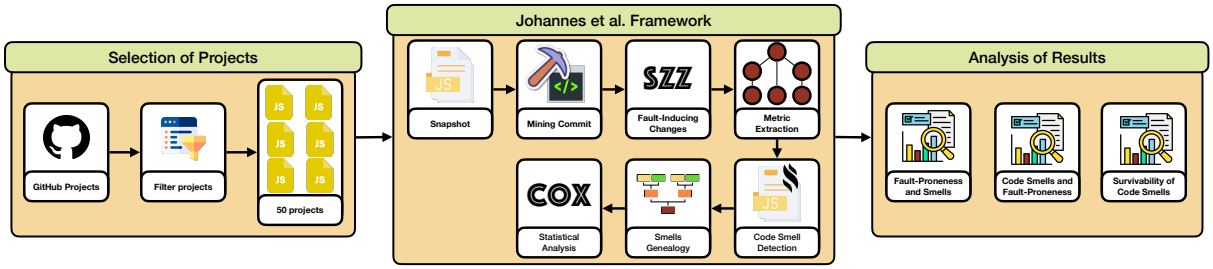


**Figure 1: Research Method Overview.**

given two commits, $C_1$ and $C_2$, on a file $F$, if a smell appears in $C_2$ but is absent in $C_1$, then $C_2$ is marked as the smell-introducing commit. Conversely, if a smell is present in $C_1$ but not in $C_2$, $C_2$ is marked as the smell-removal commit. If a smell is never removed, the framework assumes that it persists in the project.

To assess similarity between smells, the framework considers two factors: (1) whether the smell categories match, and (2) a textual similarity score between descriptions. If the categories match, a similarity score ranging from 0 to 1 is computed. If this score exceeds a predefined threshold, the smells are considered equivalent.

## 5 ANALYSIS OF THE RESULTS

This section describes the main results of our research questions (**RQs**). At the end of each research question, we also perform a comparison of our findings with those provided by Johanes *et al.* [18].

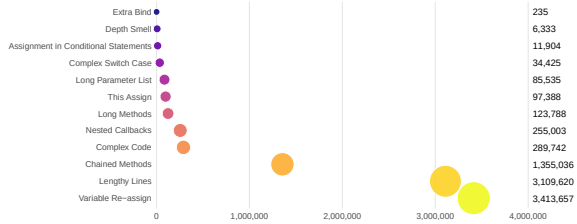


**Figure 2: Diffusion of Code Smells in JavaScript Systems. The X-Axis Reports The Number Of Occurrences, While Bubble Size Is Proportional To The Number Of Code Smells.**

Before analyzing the results, we provide some preliminary considerations regarding the diffusion of smells into JavaScript systems. Figure 2 illustrates the diffusion of code smells into the analyzed projects. The three most dominant smells are *Variable Re-Assignment*, *Lengthy Lines*, and *Chained Methods*, respectively, potentially indicating that stylistic and structural issues are particularly prevalent across the analyzed codebases. Other smells such as *Complex Code*, *Nested Callbacks*, and *Long Methods* appear with significantly lower frequency. These patterns suggest that while deeper structural issues exist, they are less common compared to more surface-level readability concerns. At the lower end of the spectrum, smells like *Assignment in Conditional Statements*, *Depth Smell*, and *Extra Bind* are relatively rare, potentially reflecting either better developer awareness of these issues or limited applicability depending on the programming paradigm used.

### 5.1 $RQ_1$. On the Hazard Ratio between Smelly and not-Smelly Files

Concerning $\mathbf{RQ_1}$, the results reveal three distinct patterns. Across 18 projects, we observe that files affected by a code smell exhibit a higher probability of long-term survival than their non-smelly counterparts. In contrast, 21 projects display the opposite behavior: smelly files are more likely to fail or be removed earlier, suggesting lower survivability and possibly higher maintenance effort. In the remaining 11 projects, no substantial difference in survivability in terms of fault is observed between smelly and non-smelly files, indicating a similar evolutionary behavior. In other words, in these cases, the presence of a code smell does not seem to significantly affect the long-term stability of the file, suggesting that other factors—such as the file's role in the system or the development practices adopted—may play a more relevant role in determining its persistence over time.

These findings partially support the hypothesis that code smells can be associated with a higher fault-proneness. However, the variation across projects highlights that the impact of code smells on survivability is not consistent and may depend on several contextual factors, such as the type of smell, system architecture, team practices, or project lifecycle. In particular, it emerges that the relationship between the presence of code smells and software degradation is far from uniform: while in some systems smells seem to accelerate the decay process, in others they may persist without leading to observable negative effects. This suggests that the influence of code smells is likely mediated by the specific characteristics of each project, including how developers handle maintenance activities, the overall quality culture within the team, and the maturity of the codebase itself.

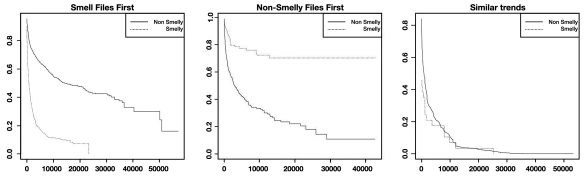


**Figure 3: Survival Curves of Lines of Code with Code Smell Compared to Lines without Code Smell.**

Figure 3 illustrates representative examples of each of the three observed trends. The first plot ("Smell Files First") shows that lines of code within smelly files tend to decay faster than those without

smells. The second plot ("Non-Smelly Files First") presents the opposite pattern, where smelly lines survive longer. Finally, the third plot ("Similar trend") shows comparable decay rates for smelly and non-smelly lines, suggesting minimal difference in survival between the two. The full set of survival plots, along with project-level classifications and detailed statistical results, is available in our publicly accessible replication package [24].

**Table 3: Hazard Ratio and Significance per project without dependencies.**

| Project | Risk Coef. | Sig. | Project | Risk Coef. | Sig. |
|---|---|---|---|---|---|
| JSDoc | 2,5 | | ExcelJS | 0,1 | |
| Axios | 0,16 | | Chrome Ext. Sam. | 0,2 | |
| Flux | 0,2 | | Blockly Samples | 0,7 | |
| Highlight.js | 0,05 | | Handlebars.js | 0,2 | |
| Jasmine | 2 | | Prism | 0,3 | |
| Async | 0,2 | | Karma | 0,2 | |
| Browserify | 0,1 | | JQueryValidation | 0,9 | |
| Emotion | 1.1 | | Forever | 0,1 | |
| Gulp | 0,2 | | Validatorjs | 0,2 | |
| LocalForage | 1 | | Markdown | 7,7 | |
| Popmotion | 0 | | Stf | 0,4 | |
| Reactstrap | 0,4 | | GpuJS | 0,1 | |
| Sweetalert2 | 0,1 | | Nodemailer | 0,1 | |
| Fetch | 0,2 | | Anime | 0 | |
| Standard | 0,3 | | Sortable | 0,1 | |
| Winston | 0 | | WS | 0 | |
| Yargs | 0 | | Piskel | 0,2 | |
| Katex | 0,1 | | Prepack | 0,8 | |
| Ungit | 1 | | Recompose | 0,4 | |
| Places | 1 | | Just | 0 | |
| Razzle | 0,4 | | Bpmn.js | 1,5 | |
| Artillery | 0,1 | | Gridsome | 0,7 | |
| Parsley.s | 0,5 | | NLP.js | 5,5 | |
| Aura | 0,9 | | VisBug | 1 | |
| A Dark Room | 0,1 | | Diagram.js | 3,9 | |

*Note:* colored cells indicate statistical significance $p < 0.05$.

Table 3 reports, for each project, the *Hazard Ratio*, and the *p-value*, expressed on a green scale. Color green indicates p-value $< 0.05$ and the significance code from the *Cox Proportional Hazards Model* contrasting smelly vs. clean files without considering dependencies. The Table shows a predominance of statistically significant effects (mostly at the 1–5% levels), alongside a non-trivial minority of non-significant cases (e.g., jQuery Validation, Emotion, LocalForage, Ungit, Places), highlighting that the smell–fault association is not universal across repositories.

Effect sizes vary markedly across projects. Several systems exhibit a large Hazard Ratio $> 1$ (e.g., JSDoc 2.539; Jasmine 2.027; Markdown 7.686; Bpmn-js 1.577; NLPjs 5.541), consistent with substantially elevated hazard in smelly files. Conversely, other projects display Hazard Ratio $< 1$ with strong significance (e.g., ExcelJS 0.10; Axios 0.156; Karma 0.228; Winston 0.042), indicating contexts where clean files experience higher hazard under the model specification. This dispersion in both sign and magnitude suggests project-specific moderators (codebase scale, practices, domain) that shape the operational meaning of "smelly" in practice.

Aggregating per-project estimates, the mean Hazard Ratio is 0.758.[4] Interpreted by inversion to express the effect for smelly files, this corresponds to ≈ 32% higher hazard (1/0.758 - 1), aligning closely with Johannes *et al.* [18], who report a 33% increase.

Table 4 shows the *hazard ratio* with dependencies. The mean Hazard Ratio is 1.588 (79.4/50), which, when inverted, corresponds to an average 37% lower hazard for smelly files. This means that, on average, LOC in files affected by a code smell are less likely to be removed or modified than those in non-smelly files.

Compared with the 45% increase reported by Johannes et al. [18], our results suggest a slightly weaker association between the presence of code smells and increased maintenance risk.

**Table 4: Hazard Ratio and Significance per project with dependencies.**

| Project | Risk Coef. | Sig. | Project | Risk Coef. | Sig. |
|---|---|---|---|---|---|
| JSDoc | 1 | | ExcelJS | 1,5 | |
| Axios | 1 | | Chrome Ext. Sam. | 2,2 | |
| Flux | 1,3 | | Blockly Samples | 1 | |
| Highlight.js | 1 | | Handlebars.js | 1 | |
| Jasmine | 1 | | Prism | 1 | |
| Async | 1 | | Karma-runner | 1 | |
| Browserify | 1 | | JQueryValidation | 1 | |
| Emotion | 1 | | Forever | 1 | |
| Gulp | 1,1 | | Validator.js | 1 | |
| Localforage | 1 | | Markdownit | 7 | |
| Popmotion | 1,3 | | Stf | 7,3 | |
| Reactstrap | 1 | | GpuJS | 1 | |
| Sweetalert | 1 | | Nodemailer | 2,5 | |
| Fetch | 1 | | Anime | 1 | |
| Standard | 1 | | Sortable | 1 | |
| Winston | 1 | | Ws | 1 | |
| Yargs | 1 | | Piskel | 1,1 | |
| Katex | 1 | | Prepack | 1 | |
| Ungit | 1 | | Recompose | 1,1 | |
| Places | 1 | | Just | 13,1 | |
| Razzle | 1 | | Bpmn-js | 1 | |
| Artillery | 1,1 | | Gridsome | 1 | |
| Parsleyjs | 1 | | NLP.js | 1,2 | |
| Aura | 1 | | VisBug | 1 | |
| A Dark Room | 1,6 | | Diagram.js | 1 | |

*Note:* colored cells indicate statistical significance $p < 0.05$.

**Table 5: Comparison of RQ1 results: Fault-proneness of smelly vs. clean files.**

| Aspect | Johannes et al. (2019) | Our replication |
|---|---|---|
| **Findings** | Smelly files show at least a 33% higher hazard rate, increasing up to 45% when dependencies are considered. | Smelly files exhibit a 32% higher hazard rate, rising to 37% with dependencies. |
| **Similarities** | Both studies confirm that smelly files are significantly more fault-prone than clean ones. | |
| **Differences** | Baseline risk 33% (up to 45% with dependencies); analysis based on 15 projects. | Similar baseline (32% vs. 33%); weaker effect with dependencies (37% vs. 45%); broader dataset (50 vs. 15 projects). |

Table 5 shows the results of $\mathbf{RQ}_1$ by comparing our replication with the findings of Johannes *et al.* [18], highlighting the main results, commonalities, and differences regarding the fault-proneness of smelly versus clean files. Both studies consistently indicate that the presence of smells substantially increases the likelihood of

[4]The arithmetic mean of the 50 per-project coefficients, 37.91/50 = 0.758.

faults, with our replication revealing an even stronger effect when accounting for dependencies.

## 5.2 $RQ_2$. On the fault-proneness in files containing smells

Figure 4 depicts the distribution of the Hazard Ratio for each code smell across the analyzed projects. Overall, most smells exhibit median values in the range of 0.2–0.3, confirming a general but moderate association with fault proneness. However, the dispersion varies substantially: *Variable Re-Assignment*, *Lengthy Lines*, and *This Assign* show wider interquartile ranges and long upper whiskers, indicating that in some projects these smells are linked to considerably higher risks. In contrast, *Extra Bind* and *Depth Smell* exhibit narrow interquartile ranges and limited spread, suggesting a weaker and more stable relationship across projects. Interestingly, *Nested Callbacks* and *Long Methods*, often cited as critical design flaws, display consistently elevated medians, reinforcing their role as predictors of faults. The variability observed for other smells (e.g., *Complex Switch* and *Assign in Condition*) suggests a more context-dependent impact, which may depend on project-specific practices or architectural choices. These patterns highlight that while some smells systematically increase fault hazard, others exert their effect only under certain conditions, pointing to the importance of project-aware prioritization in smell detection and refactoring.

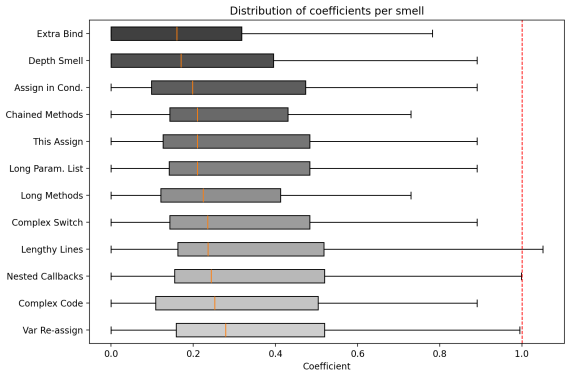


**Figure 4: Hazard Ratio for Smells. Bars Show The Distribution Per Smell, With Median And Interquartile Range.**

**Table 6: Risk Rate Associated with Each Type of Code Smell.**

| Smell | Prevalence |
|---|---|
| Variable Re-Assignment | 34,60% |
| Complex Code | 31,40% |
| Assignment in Conditional Statements | 30,00% |
| Nested Callbacks | 26,50% |
| Chained Methods | 26,50% |
| Long Parameter List | 23,75% |
| Lengthy Lines | 21,75% |
| Long Methods | 20,50% |
| Extra Bind | 20,00% |
| This Assign | 19,50% |
| Complex Switch Case | 17,60% |
| Depth Smell | 16,25% |

Table 6 shows the risk rate for each smell. The highest risk rates are for *Variable Re-Assignment* (34,60%), *Complex Code* (31,40%), and *Assignment in Conditional Statements* (30,00%). A second tier includes *Nested Callbacks* and *Chained Methods* (both 26,50%), followed by *Long Parameter List* (23,75%), *Lengthy Lines* (21,75%), and *Long Methods* (20,50%). These findings align with the results reported by Johannes *et al.* [18], who identified *Variable Re-Assignment* as the most critical smell, associated with a 34.60% increase in the likelihood of fault introduction. Similarly, *Complex Code* and *Assignment in Conditional Statements* were linked to elevated fault risks of 31.40% and 30.00%, respectively, reinforcing their relevance as indicators of error-proneness and maintenance difficulties. Smells such as *Depth Smell* and *Complex Switch Case* were found to have lower associated risk levels, with average fault probabilities of 16.25% and 17.60%. Although these smells may still affect code readability and maintainability, their actual contribution to fault occurrence appears notably less severe compared to the more impactful smells.

**Table 7: Comparison of $RQ_2$ results: Equality of code smells in fault-proneness.**

| Aspect | Johannes et al. (2019) | Our replication |
|---|---|---|
| **Findings** | Identifies three most critical smells: Variable Re-assignment, Assignment in Conditional Statements, and Complex Code. | Confirms the same three smells as the most critical, but shows that not all smells are equally harmful. |
| **Similarities** | Both studies converge on the same three smells as the most fault-prone | |
| **Differences** | Emphasizes a fixed set of three critical smells. | Finds that only a subset is consistently harmful, while others (e.g., Long Method, Nested Callbacks) exhibit variable impact across projects. |

Table 7 summarizes the results of $\mathbf{RQ_2}$ and compares our findings with the work of Johannes *et al.* [18] in terms of the equality of code smells in fault-proneness. It highlights the main findings, similarities, and differences regarding which smells are most harmful and whether their impact is consistent across contexts. Overall, while both studies identify the same critical smells, our replication nuances these findings by revealing that the fault-proneness of smells is context-dependent rather than uniform.

## 5.3 $RQ_3$. On the Smells Survivability in JavaScript Systems

Table 8 overviews of smell survivability across the projects, reporting the percentage of smells removed or still active at the time of the last analyzed commit, along with the average survivability of smells expressed in both days and commits.

Based on the percentage of smells removed, the projects can be grouped into three categories: those with a high removal rate (over 70%), such as *JSDoc*, *ExcelJS*, and *Axios*, totaling 32 projects; those with a low removal rate (below 45%), including *Handlebars.js* and *Prepack*, comprising 10 projects; and those with a moderate removal rate (between 45% and 70%), such as *Piskel* and *Gridsome*, accounting for 8 projects.

The survivability values reveal notable differences in how long smells persist across codebases. On the one hand, projects such as

*Handlebars.js* and *Prism* exhibit some of the highest mean survivability—exceeding 3,000 days (≈ 8 years) and over 1,000 commits. This suggests that smells in these systems are not only tolerated, but may become deeply embedded in the codebase, possibly due to architectural inertia, limited refactoring, or a perception that they are not harmful enough to warrant prompt removal.

On the other hand, projects like *Sweetalert2*, *Just*, and *Ungit* exhibit extremely low survivability—sometimes under 100 days or commits—indicating the need for more proactive maintenance, where issues are addressed promptly. This may reflect agile development practices or a stronger emphasis on quality assurance.

Interestingly, projects with moderate removal rates exhibit mixed survivability trends. For instance, *Piskel* and *Gridsome* show relatively high survivability in both time and number of commits, suggesting that although smells are eventually removed, they often persist for a considerable period.

**Table 8: Survivability and Status of Entities per Project.**

| Project | % Removed | % Active | Mean Survivability (#Days) | Mean Survivability (#Commits) | % Smells from Day One |
|---|---|---|---|---|---|
| JSDoc | 99,3% | 0,7% | 425 | 424 | 59,6% |
| ExcelJS | 97,9% | 2,1% | 618 | 342 | 62,2% |
| Axios | 86,4% | 13,6% | 599 | 204 | 16,4% |
| Chrome Ext, Samples | 9,6% | 90,4% | 3228 | 1048 | 95,5% |
| Flux | 72,5% | 27,5% | 604 | 115 | 70,3% |
| Blockly Samples | 41,8% | 58,2% | 776 | 725 | 81,1% |
| Highlight.js | 99,4% | 0,6% | 825 | 759 | 56,2% |
| Handlebars.js | 39,5% | 60,5% | 3350 | 1212 | 53,7% |
| Jasmine | 96,1% | 3,9% | 813 | 373 | 42,4% |
| Prism | 27,5% | 72,5% | 1448 | 1329 | 51,2% |
| Async | 74,4% | 25,6% | 989 | 351 | 50,1% |
| Karma | 92,9% | 7,1% | 441 | 504 | 81,2% |
| Browserify | 78,8% | 21,2% | 282 | 504 | 36,5% |
| JQuery Validation | 50,4% | 49,6% | 2264 | 303 | 78,4% |
| Emotion | 88,4% | 11,6% | 413 | 220 | 51,2% |
| Forever | 76% | 24% | 755 | 160 | 38% |
| Gulp | 66,7% | 33,3% | 520 | 207 | 23,7% |
| Validator.js | 100% | 0% | 394 | 159 | 30,6% |
| LocalForage | 76,5% | 23,5% | 1005 | 200 | 71,7% |
| Markdown | 100% | 0% | 1201 | 328 | 34,7% |
| Popmotion | 93,4% | 6,5% | 170 | 155 | 59,9% |
| Stf | 45% | 55% | 1085 | 891 | 41,3% |
| Reactstrap | 97,3% | 2,6% | 504 | 261 | 94,6% |
| Gpu.js | 98,8% | 1,2% | 319 | 269 | 51,3% |
| Sweetalert2 | 99,6% | 0,4% | 122 | 155 | 50,3% |
| Nodemailer | 99,8% | 0,1% | 285 | 84 | 49,4% |
| Fetch | 47,8% | 52,2% | 829 | 154 | 8,6% |
| Anime | 26,8% | 73,2% | 1538 | 235 | 75,8% |
| Standard | 64,9% | 35,1% | 469 | 369 | 46,8% |
| Sortable | 48% | 52% | 1136 | 171 | 35,3% |
| Winston | 91,7% | 8,3% | 1007 | 384 | 38,3% |
| Ws | 100% | 0% | 919 | 399 | 26,8% |
| Yargs | 97,1% | 2,9% | 316 | 289 | 20% |
| Piskel | 60,6% | 39,4% | 1522 | 525 | 66,9% |
| Katex | 98,8% | 1,1% | 365 | 161 | 55,8% |
| Prepack | 23,1% | 76,9% | 999 | 730 | 85% |
| Ungit | 100% | 0% | 64 | 139 | 96,2% |
| Recompose | 94,1% | 5,9% | 221 | 214 | 47,1% |
| Places | 98,6% | 1,4% | 210 | 98 | 100% |
| Just | 99% | 1% | 59 | 58 | 96% |
| Razzle | 21% | 79% | 686 | 703 | 93,8% |
| Bpmn-js | 98,7% | 1,3% | 517 | 351 | 50% |
| Artillery | 44,2% | 55,8% | 469 | 512 | 83,5% |
| Gridsome | 61,5% | 38,5% | 554 | 600 | 89,7% |
| Parsley.js | 50% | 50% | 1732 | 438 | 77,5% |
| NLP.js | 4,1% | 96% | 1409 | 1479 | 99,8% |
| Aura | 97,9% | 2,1% | 202 | 109 | 79,7% |
| VisBug | 100% | 0% | 832 | 1069 | 100% |
| A Dark Room | 77,8% | 22,2% | 1067 | 226 | 31,4% |
| Diagram.js | 100% | 0% | 442 | 216 | 46,5% |

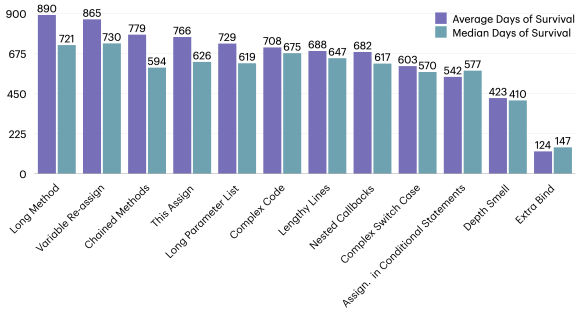


**Figure 5: Smell Survivability across Projects.**

Looking at the percentage of smells introduced at file creation (last column), we observe three distinct groups: 14 projects introduced fewer than 45% of their smells from the beginning, 17 projects introduced between 45% and 70%, and 19 projects introduced more than 70%. When cross-referencing these groups with the smell removal categories, a clear pattern emerges: projects with low removal rates tend to also have a higher proportion of smells introduced from day one (average 76%), while those with moderate or high removal rates introduce significantly fewer smells at file creation (averaging around 53–56%). This may suggest that projects with more persistent smells also suffer from initial code quality issues, which in turn may hinder long-term maintainability.

Figure 5 further illustrates how survivability varies across different types of code smells. We can observe that *Long Method* and *Variable Re-Assignment* exhibit the highest average survivability, 890 and 865 days, respectively. This suggests that developers are often reluctant or slow to refactor these issues, possibly due to their perceived harmlessness or the complexity involved in restructuring such code. At the same time, *Extra Bind* and *Depth Smell* are resolved far more quickly, surviving on average for just 124 and 423 days respectively. These smells may be more easily detected, more clearly harmful, or simpler to fix, prompting quicker action. We also observed that, while some smells like *Chained Methods* and *This Assignment* also survive for extended periods (766–779 days), their median survivability is noticeably lower than the mean, indicating that while most instances are resolved in a reasonable time, a subset persists for much longer—skewing the average upward.

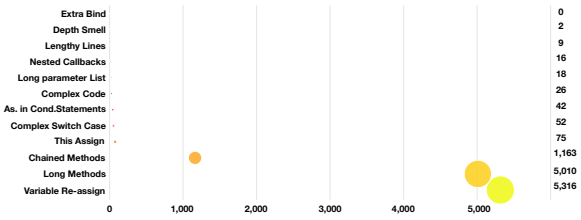


**Figure 6: Most Common Smells Introduced at File Creation. Bubble size is proportional to smell diffusion.**

Figure 6 complements these findings by showing which smells are most commonly introduced at file creation. Notably, over 94% of *Variable Re-assignment* and *Long Method* instances appear at file

creation, aligning with their high survivability and suggesting these smells are often part of the initial design and rarely revisited. In contrast, smells like *Depth Smell* and *Extra Bind*, which are resolved quickly, are almost never introduced at file creation. The results are in line with Johannes *et al.* [18] who find *Variable Re-Assignment* is the most prevalent smell introduced during file creation.

Table 9 summarizes the results of $\mathbf{RQ}_3$, comparing our findings with those reported by Johannes *et al.* [18]. Both studies confirm that code smells tend to persist once introduced, though our replication reveals substantial variation in their survivability across projects.

**Table 9: Comparison of $\mathbf{RQ}_3$ results: Survivability of code smells in JavaScript projects.**

| Aspect | Johannes et al. (2019) | Our replication |
|---|---|---|
| **Findings** | Many smells are introduced at file creation and persist long-term; Variable Re-assign emerges as the most prevalent and long-lived. | Confirms the persistence of early smells, but shows survivability varies considerably: e.g., Long Method and Variable Re-assign persist for ~900 days on average. |
| **Similarities** | Both studies agree that smells introduced early tend to persist throughout the project lifetime. | |
| **Differences** | Emphasizes the survivability of smells introduced at creation, with the prevalence of Variable Re-assign. | Highlights strong project dependency: some systems (e.g., Axios) remove smells quickly, while others (e.g., Handlebars.js) retain them for years. |

## 6 DISCUSSION AND IMPLICATIONS

We first discuss the results of our study in relation to the replicated work, highlighting differences and explanations. We then present the implications for developers, tool vendors, and researchers.

### 6.1 Comparison with the Original Study

Our study confirms several findings of the original study, while also revealing notable differences that offer new perspectives. Both studies consistently show that the presence of certain code smells, i.e., *Assignment in Conditional Statements* and *Complex Code*, is associated with an increased risk of faults. This convergence strengthens prior evidence that specific smell types may serve as reliable indicators of quality risks, even across different samples and project ecosystems. Moreover, both analyses reveal that many smells are introduced at the moment of file creation and tend to persist across subsequent releases. This pattern suggests that *early-stage design decisions have lasting implications for maintainability and fault-proneness*. These findings contribute to the body of knowledge on code smells in JavaScript by reinforcing the idea that not all smells are equal in severity and that certain patterns consistently signal technical debt with long-term effects. They also echo earlier results from studies conducted in statically typed languages, where the early introduction and long-term survivability of smells has likewise been reported [35]. This cross-language consistency implies that some smell-related risks are not merely artifacts of a particular programming paradigm or ecosystem, but may reflect more general principles of software evolution and design degradation.

However, differences also emerge. First, our replication identifies a smaller number of smells with statistically significant fault associations than the original study. While Johannes *et al.* found a broader subset of the 12 JavaScript-specific smells to be risk-inducing, our results suggest that only a limited group, namely *Assignment in Conditional Statements* and *Complex Code*, consistently correlates with fault-proneness across a more varied project landscape. This discrepancy may stem from differences in project maturity, development practices, contributor dynamics, or domain-specific idioms. For instance, Ahmed *et al.* [2] found that code quality in open-source systems is more strongly influenced by the growth in the number of contributors than by codebase size itself, suggesting that community structure may mediate how smells are introduced and addressed. These findings point to a critical insight for code smell research: *the diffuseness and severity of smells are highly context-dependent*. This challenges earlier assumptions, often implicit in foundational studies on code smells, that all smells are equally harmful or universally indicative of poor design. While prior work has explored developers' subjective perceptions of smell severity [26, 34], there is still limited empirical evidence on how intrinsic project characteristics (e.g., size, contributor churn, maturity, domain) influence the actual fault-related risks associated with smells. Our findings emphasize the need for project-aware smell assessment frameworks that account for contextual variability, rather than relying on fixed smell taxonomies or uniform severity labels.

Second, while both studies confirm that smells are frequently introduced at file inception, a finding aligned with earlier observations on smell lifespan [35], our replication reveals greater heterogeneity in their persistence and their relationship to faults. In some projects, smelly files persisted across multiple releases without increasing fault-proneness, whereas in others, such files were short-lived and closely linked to defects. This variability may reflect the influence of project-specific quality assurance mechanisms. For instance, some repositories may employ just-in-time quality assurance tools [19], AI- or LLM-assisted code reviews [36], or pre-merge validation pipelines that limit the propagation of risky changes [37, 38]. Additionally, differences in contribution guidelines [9] and enforcement practices can affect how actively smells are detected, tolerated, or refactored over time. These findings suggest that the lifecycle and impact of smells are shaped by organizational and process-level factors, not solely by their presence or structural characteristics, and highlight the importance of investigating the socio-technical context in which smells evolve, moving beyond static detection to more holistic models of smell management.

### 6.2 Implications of the Study

This replication study carries implications for developers (</>), tool vendors (🧰), and researchers (👩‍🏫). The implications are relevant given the widespread adoption of JavaScript and its growing presence in automatically generated code.

First, not all code smells are equally harmful. Among the 12 JavaScript-specific smells investigated, only a subset exhibits a statistically significant association with fault-proneness. Practitioners should therefore focus remediation efforts on high-risk smells, rather than applying uniform effort across all detected issues.

</> *Not all code smells are equally harmful; prioritizing high-risk smells (e.g., Variable Re-Assignment, Complex Code) leads to more effective quality improvements.*

Second, many smells are introduced at the earliest stages of a file's life cycle and tend to persist indefinitely. In some projects, more than 70% of smells originated at file creation, underscoring the importance of early-stage code hygiene. Preventive measures, such as CI checks or review practices, are particularly important when introducing new modules or automatically generated code.

</> *Most smells introduced during file creation persist over the long term; applying quality in the early stages is critical to reduce long-term technical debt.*

Third, the survivability and fault-proneness of smells vary across projects. In some systems, smelly files persisted without increasing fault risk, while in others they were quickly removed or strongly correlated with defects. This variability implies that smell management policies should be grounded in project-specific data, rather than guided by generic refactoring rules.

</> *Code smell behavior varies across projects; management policies must be adapted to historical and contextual evidence.*

From a tooling perspective, static analyzers could be enhanced by reporting not only rule violations but also fault correlations and survivability trends. Features such as the detection of "born-smelly" files, introduced with critical smells and rarely modified afterward, would help in identifying persistent sources of technical debt.

*Static analysis tools should communicate fault risk and survivability, supporting more informed refactoring decisions.*

Ultimately, our findings suggest avenues for future research. Since this study relies exclusively on open-source projects, replication in industrial contexts is needed to assess generalizability. Moreover, while our analysis focuses on risk, future work should consider the cost of remediation and model the trade-offs between removal effort and actual reliability or maintainability gains.

*Future research should validate these findings in industrial settings and investigate cost–benefit trade-offs in smell remediation.*

## 7 THREATS TO VALIDITY

In this section, we discuss threats to validity and the mitigation strategies we applied [39].

*Construct Validity.* Following Johannes *et al.* [18], we estimated prior faults by identifying fault-fixing commits through keyword-based mining of commit logs (e.g., FIX, #, GH-) and bug IDs [11]. Although widely adopted and validated in prior work (e.g., [17, 30]), this heuristic may miss fixes due to incomplete or inconsistently formatted messages.

The SZZ algorithm used to trace fault-inducing commits is also imperfect. To mitigate inaccuracies, we applied the refinements proposed by Da Costa [6], excluding commits that modify only blank or comment lines and those temporally distant from the associated issue. As in the original study, we restricted the analysis to the master branch, acknowledging that cross-branch timing differences may introduce minor imprecision.

For reconstructing smell genealogies, we adopted the same 70% similarity threshold as in the original work [18], which was selected via sensitivity analysis. While this threshold may occasionally merge distinct smells or separate similar ones, it ensures methodological consistency with the baseline study.

*Internal Validity.* We adopted the same metric approach as the original study, relying on the AST generated by ESLint. Accordingly, our results depend on the accuracy of ESLint's analysis, which we consider sufficiently reliable. Unlike the original release-level analysis, we operate at the commit level, enabling finer-grained tracking of code smell and fault evolution. However, this choice may introduce additional variability, as commits can reflect transient changes or refactorings that would be consolidated in releases. Thus, some observed modifications or removals may capture only commit practices rather than the intrinsic properties of smelly code. As in the original work, we applied a logarithmic link function for selected covariates in the survival models. Although alternative link functions might improve fit for specific variables, non-proportionality tests indicate that the models remain appropriate for our dataset.

*External Validity.* We analyzed 50 open-source JavaScript projects and over 90,000 commits. While these projects span diverse domains and vary in size, the scope remains limited. Therefore, additional validation across a broader range of JavaScript systems and a more comprehensive set of code-smell types would be beneficial.

*Conclusion Validity.* We relied on the *Cox Proportional Hazards Model* to assess the relationship between code smells and fault-proneness. While this model is well-established and suitable for survival analysis, its conclusions depend heavily on correct specification and the validity of underlying assumptions. We performed non-proportionality tests to verify model fit and ensure that the assumptions were adequately met.

## 8 CONCLUSIONS

This work is a replication of the study of Johannes *et al.* [18], which examined the relationship between smells and fault-proneness in JavaScript systems. By analyzing 50 open-source projects, we confirmed that smelly files are more fault-prone, with *Variable Re-Assignment*, *Complex Code*, and *Assignment in Conditional Statements* emerging as particularly harmful. Our results revealed that smell persistence varies substantially across projects, and that those introduced at file creation often persist over time.

This paper provides three contributions: (i) A large-scale empirical investigation of smells in JavaScript projects, highlighting their persistence and association with fault-proneness; (ii) An extension of the methodology by Johannes *et al.* [18], enhancing robustness and enabling a more fine-grained evaluation across a broader project set; and (iii) A publicly available replication package with all data, scripts, and results to support reproducibility [24].

In the future, we plan to analyze a wider range of JavaScript projects and complement quantitative analyses with developer studies. Our long-term goal is to design context-aware approaches for detecting and managing smells, thereby supporting more effective maintenance and refactoring practices.

## ACKNOWLEDGMENT

Valeria acknowledges the support of the European HORIZON-KDT-JU-2023-2-RIA research project MATISSE (grant 101140216-2, KDT232RIA_00017).

# REFERENCES


[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 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*. IEEE, 181–190.

[2] Iftekhar Ahmed, Soroush Ghorashi, and Carlos Jensen. 2014. An exploration of code quality in FOSS projects. In *IFIP International Conference on Open Source Systems*. Springer, 181–190.

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

[4] Sufyan Bin Uzayr, Nicholas Cloud, and Tim Ambler. 2019. *JavaScript Frameworks for Modern Web Development*. Springer.

[5] Douglas Crockford. 2011. Jslint: The javascript code quality tool. *URL http://www. jslint. com* 95 (2011).

[6] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.

[7] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in github for MSR studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 560–564.

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

[9] Omar Elazhary, Margaret-Anne Storey, Neil Ernst, and Andy Zaidman. 2019. Do as i do, not as i say: Do contribution guidelines match the github contribution process?. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 286–290.

[10] Amin Milani Fard and Ali Mesbah. 2013. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 116–125.

[11] Michael Fischer, Martin Pinzger, and Harald C. Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 23.

[12] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[13] John Fox and Sanford Weisberg. 2002. Cox proportional-hazards regression for survival data. *An R and S-PLUS companion to applied regression* 2002 (2002).

[14] Giammaria Giordano, Antonio Fasulo, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. 2022. On the Evolution of Inheritance and Delegation Mechanisms and Their Impact on Code Quality. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 947–958. https://doi.org/10.1109/SANER53432.2022.00113

[15] Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, and Filomena Ferrucci. 2023. The Yin and Yang of Software Quality: On the Relationship between Design Patterns and Code Smells. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 227–234. https://doi.org/10.1109/SEAA60479.2023.00043

[16] Aakanshi Gupta, Bharti Suri, and Sanjay Misra. 2017. A systematic literature review: code bad smells in java source code. In *International Conference on Computational Science and Its Applications*. Springer, 665–682.

[17] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. 2013. Mining the relationship between anti-patterns dependencies and fault-proneness. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 351–360. https://doi.org/10.1109/WCRE.2013.6671310

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

[19] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[20] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[21] Danyu Y Lin and Lee-Jen Wei. 1989. The robust inference for the Cox proportional hazards model. *Journal of the American statistical Association* 84, 408 (1989), 1074–1078.

[22] J.C. Munson and S.G. Elbaum. 1998. Code churn: a measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 24–31. https://doi.org/10.1109/ICSM.1998.738486

[23] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2012. Detection of embedded code smells in dynamic web applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 282–285.

[24] Kevin Pacifico, Giammaria Giordano, Valeria Pontillo, Massimiliano Di Penta, Damian A. Tamburri, and Fabio Palomba. 2025. Replication Package of "An Empirical Analysis on the Relationship between Code Smells and Fault-Proneness In JavaScript Projects: A Replication Study". https://giammariagiordano.github.io/javascript_code_smells_and_faults. Accessed: 2025-08-02.

[25] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. In *Proceedings of the 40th International Conference on Software Engineering*. 482–482.

[26] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 101–110.

[27] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 155–165.

[28] Amir Saboury, Pooya Musavi, Foutse Khomh, and Giulio Antoniol. 2017. An empirical study of code smells in javascript projects. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 294–305.

[29] Raed Shatnawi and Wei Li. 2006. An investigation of bad smells in object-oriented design. In *Third International Conference on Information Technology: New Generations (ITNG'06)*. IEEE, 161–165.

[30] E SHIHAB. 2013. Studying Re-Opened Bugs in Open Source Software, Empirical Software Engineering. *http://link. springer. com/article/10.1007% 2Fs10664-012-9228-6* (2013).

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

[32] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[33] Statista Research Department. 2023. Most Used Programming Languages Among Developers Worldwide. Accessed: 2025-08-02. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

[34] Davide Taibi, Andrea Janes, and Valentina Lenarduzzi. 2017. How developers perceive smells in source code: A replicated study. *Information and Software Technology* 92 (2017), 223–235.

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

[36] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling chatgpt's usage in open source projects: A mining-based study. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 571–583.

[37] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 805–816.

[38] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C Gall. 2018. Continuous code quality: Are we (really) doing that?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 790–795.

[39] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[40] Morteza Zakeri-Nasrabadi, Saeed Parsa, Ehsan Esmaili, and Fabio Palomba. 2023. A systematic literature review on the code smells datasets and validation mechanisms. *Comput. Surveys* 55, 13s (2023), 1–48.