Improving JavaScript Test Quality with Large Language Models: Lessons from Test Smell Refactoring

Gabriel Amaral
UEFS
Feira de Santana, Bahia, Brazil
gabrielamaralsousa@gmail.com

Henrique Gomes
UFMG
Belo Horizonte, Minas Gerais, Brazil
henrique.mg.bh@gmail.com

Eduardo Figueiredo
UFMG
Belo Horizonte, Minas Gerais, Brazil
figueiredo@dcc.ufmg.br

Carla Bezerra
UFC
Quixadá, Bahia, Brazil
carlailane@ufc.br

Larissa Rocha
UNEB/PGCC-UEFS
Alagoinhas, Bahia, Brazil
larissabastos@uneb.br

ABSTRACT

Test smells—poor design choices in test code—can hinder test maintainability, clarity, and reliability. Prior studies have proposed rule-based detection tools and manual refactoring strategies, most focus on statically typed languages such as Java. In this paper, we investigate the potential of Large Language Models (LLMs) to automatically refactor test smells in JavaScript, a dynamically typed and widely used language with limited prior research in this area. We conducted an empirical study using GitHub Copilot Chat and Amazon CodeWhisperer to refactor 148 test smell instances across 10 real-world JavaScript projects. Our evaluation assessed smell removal effectiveness, behavioral preservation, introduction of new smells, and structural code quality based on six software metrics. Results show that Copilot removed 58.78% of the smells successfully, outperforming Whisperer's 47.30%, while both tools preserved test behavior in most cases. However, both also introduced new smells, highlighting current limitations. Our findings reveal the strengths and trade-offs of LLM-based refactoring and provide insights for building more reliable and smell-aware testing tools for JavaScript.

KEYWORDS

Test Smells, Large Language Models (LLMs), JavaScript

1 Introduction

Testing is a critical phase in the software development lifecycle, ensuring reliability and quality while reducing long-term costs through early defect detection [3, 5, 30]. Although automated testing offers scalability and efficiency over manual methods [2], it brings challenges related to test code quality, notably the presence of *test smells*—poor design practices that compromise maintainability, readability, and effectiveness [22, 26, 36]. Smells such as code duplication, unclear logic, inter-test dependencies, and overly complex or rigid cases hinder adaptation to system changes and increase maintenance, as minor updates may require extensive test rework [36]. Addressing them is essential, as empirical studies reveal their widespread impact on test suite sustainability and their role in undermining confidence in automated testing [26, 36].

Despite these advancements, most studies and tools in this domain have concentrated on statically typed languages, such as Java [38], Scala [10], Smalltalk [31], and C++ [4]. In contrast, JavaScript, despite being one of the most widely used programming languages [6], has seen limited research in automated test smell refactoring, with most existing work focused on detection only. For example, a systematic mapping study revealed 22 tools for detecting test smells, yet none were specifically designed for JavaScript codebases [1]. This gap is particularly notable given JavaScript's dynamic nature and widespread adoption. More recent efforts have started to explore this area by investigating smell detection in JavaScript test code [17, 25, 33].

In addition to detection, recent research has also focused on automating or semi-automating the refactoring of test smells [28, 29, 32, 34]. With the rise of generative AI, researchers have been exploring the use of language models for such tasks [21]. Large Language Models (LLMs) have emerged as promising tools for automating various software engineering activities, including test code improvements [16, 42]. However, their application to test refactoring is still in its early stages and faces key challenges, such as hallucinated code, inconsistent refactorings, and limited understanding of testing frameworks. Even so, LLM-powered tools, like Copilot Chat [13], have demonstrated impressive performance in generating and refactoring code, with studies reporting high levels of accuracy in these tasks [7, 43]. More recently, investigations have begun to assess how LLMs perform specifically in the context of test smell detection and refactoring [8]. However, as far as we are concerned, no previous work has explored the use of LLMs to refactor test smells in JavaScript.

This study aims to evaluate the effectiveness of LLMs in refactoring JavaScript test code affected by test smells. We conducted an empirical study involving two LLMs — GitHub Copilot Chat and Amazon CodeWhisperer — initially using a zero-shot prompting approach to refactor test code snippets containing known test smells. These test smells were identified in open-source JavaScript projects using two automated detection tools. The study targeted a subset of ten test smells, such as *Conditional Test Logic*, *Overcommented Test*, and *Suboptimal Assertion*. We analyzed a total of 148 instances of test smells extracted from 10 open-source Javascript projects.

To assess the effectiveness of LLM-generated refactorings, we developed a framework evaluating test result changes, coverage variation, smell removal and introduction, and software quality metrics for long-term maintainability. Both LLMs produced syntactically valid refactorings, though they differed in grasping testing idioms. Copilot outperformed Whisperer, removing 58.78% of smells (87/148), especially in *Conditional Test Logic*, likely due to stronger

contextual inference. Whisperer removed 47.30% (70/148). Quality metrics showed both preserved core properties like complexity and maintainability, using distinct strategies: Copilot applied conservative, cognitively efficient changes; Whisperer prioritized structural optimization, improving density but slightly increasing code size. Despite differences, both maintained high maintainability, suggesting complementary strengths in AI-assisted refactoring and emphasizing the need for further domain-specific tuning.

2 Background and Related Work

2.1 Tools for Test Smell Detection

Test smells arise from poor design decisions in test code and can hinder maintainability and reliability [26]. For instance, *Sensitive Equality* makes tests brittle by comparing objects via `toString`, while *Assertion Roulette* obscures test failures due to multiple undocumented assertions [36]. Detecting and resolving such issues is particularly challenging in large test suites, motivating the development of automated detection tools [19, 27, 31, 32, 37, 38].

Early tools focused on statically typed languages. For example, TestLint [31] identifies maintainability problems in Smalltalk tests, while TeCReVis [19] visualizes redundancy in Java test coverage. In the Java ecosystem, tsDetect [27] uses AST-based rules to detect 19 test smells in JUnit with high precision and recall. Tools like JNose [37, 38] and RAIDE [32] extend tsDetect's capabilities by incorporating smell-specific metrics and IDE-integrated refactoring support.

JavaScript, however, has only recently received attention. Steel [17] detects 16 JavaScript-specific test smells, such as *Eager Test*, *Lazy Test*, and *Assertion Roulette*. TestSmellDetectorJs [33] builds on Steel to incorporate eight additional smells, adapting heuristics from PyNose [39]. More recently, SNUTS.JS [25] introduced detection for 15 diverse smells, including *Overcommented Test*, *General Fixture*, and *Transcripting Test*.

While these tools are rule-based and limited by static patterns, our study investigates whether LLMs can offer more flexible and context-aware support for smell resolution through automated refactoring.

2.2 LLMs for Tests

Large Language Models (LLMs) are transforming Software Engineering (SE) tasks, including code generation [23] and bug repair [24]. However, their role in testing remains less explored. Recent studies have assessed LLMs' ability to generate test code [15], detect test smells [20], and refactor them [11, 12].

Hasan et al. [15] showed that GPT-4o, Gemini, Llama 3.1, and Mistral can generate readable and accurate tests from use cases. Lucas et al. [20] evaluated smell detection using ChatGPT-4, Gemini Advanced, and Mistral Large—ChatGPT-4 identified 21 of 30 smell types, outperforming the others. In the refactoring domain, Gao et al. [12] proposed UTRefactor, a Java-based LLM framework that applied chain-of-thought prompting and achieved 89% smell reduction in 6 repositories. Fatima et al. [11] explored LLMs in repairing flaky tests using a curated dataset and improved ChatGPT 3.5's repair rate to 51–83%.

Our contribution differs in three key ways. First, we focus on JavaScript, a dynamically typed language with limited prior work on smell refactoring. Second, we evaluate two production-grade LLM-based assistants—Copilot Chat and CodeWhisperer—on real-world test suites. Third, we go beyond detection by analyzing removal effectiveness, behavioral preservation, metric-based quality, and the emergence of new smells after refactoring.

3 Study Design

3.1 Goals and Research Questions

The primary goal of this paper is to evaluate the effectiveness of Large Language Models (LLMs) in automated test code refactoring, with a particular focus on mitigating test smells.

To guide this investigation, we formulated four research questions. **RQ1** examines to what extent LLMs can remove test smells from JavaScript test code without compromising test behavior or coverage. **RQ2** explores whether LLM-assisted refactoring introduces new test smells, including combinations that may degrade code quality. **RQ3** compares the effectiveness of different LLMs in removing specific types of test smells, identifying tool-specific strengths and limitations. Finally, **RQ4** analyzes how LLM-assisted refactoring impacts the structural quality of test code, based on changes in software metrics such as complexity, size, and maintainability.

The study follows a structured, multi-step process as presented in Figure 1. This study is split into three phases: pre-refactoring activities, refactoring process, and post-refactoring analysis. In the pre-refactoring phase, we selected the test smells detection tools, the test smells under investigation, the repositories used in this study, and the LLM-based tools. In the second phase, we executed the refactorings based on pre-defined prompts. Finally, the post-refactoring analysis consisted of running the test smells tools detection again in the repositories refactored to verify if the LLM-based tools removed the smells in the refactored code, as well as analysing the results. These steps, are better detailed in the following subsections.

3.2 Pre-refactoring Activities

In the first phase, we selected the following: (i) test smells detection tools, (ii) test smells under investigation, (iii) repositories used in this study, and (iv) LLM-based tools.

3.2.1 Selecting test smells detection tools. After conducting an ad-hoc review of the literature on JavaScript test smell detection tools, we identified three main tools: Steel [17], TestSmellDetectorJs [33], and SNUTS.JS [25]. To determine the most suitable tools for this study, we conducted a hands-on evaluation of each one. During this process, we found that *TestSmellDetectorJs* did not provide precise location details for the identified smells—reporting only their presence in a given file. Then, we opted not to include this tool in the final analysis. Based on these considerations, we selected Steel and SNUTS.JS for use in this study. Both tools demonstrated the ability to detect a wide range of test smells and, importantly, offered detailed information about their location in the code—an essential aspect for the subsequent analysis and refactoring activities.

3.2.2 Selecting test smells. The selection of test smells was guided by the classifications provided by each detection tool. Notably, there was minimal overlap between the smells supported by SNUTS.JS

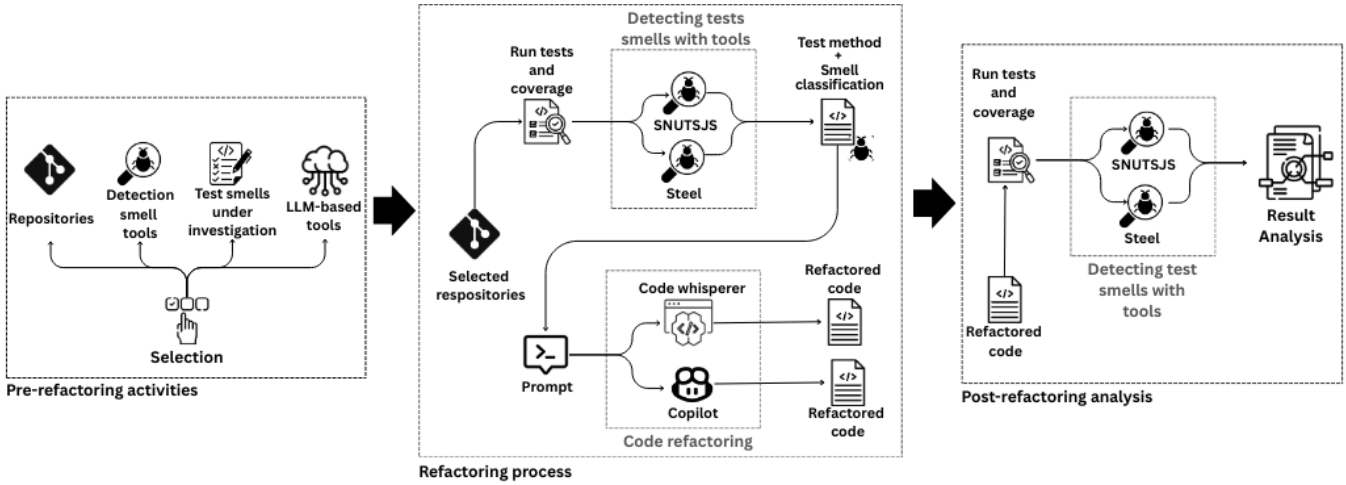

Figure 1: Workflow of the Design

and Steel, with only five smells in common, which led us to prioritize distinct smells from each tool. This decision aimed to increase the diversity of test smell types analyzed, thereby enhancing the representativeness of our study across different categories of design issues in test code.

Beyond selecting tool-specific smells, we applied inclusion criteria to ensure the smells were actionable within this study's scope. We focused on smells occurring within test method bodies, as these were the primary refactoring targets. Smells extending beyond this scope—such as *General Fixture* (overly generic setups) or *Global Variable* (shared state across the suite)—were excluded due to the difficulty of consistently and automatically refactoring them. We also discarded smells lacking executable code, which offer no context for refactoring, such as *Comments Only Test*, where the entire test or block is commented out.

Based on these considerations, we selected ten test smells for this study. From the SNUTS.JS tool, we included *Conditional Test Logic*, *Overcommented Test*, *Suboptimal Assert*, *Test Without Description*, and *Sensitive Equality*. From the Steel tool, we selected *Assertion Roulette*, *Duplicate Assert*, *Magic Number*, *Lazy Test*, and *Redundant Print*. Together, these smells span both assertion-level and structural-level issues, offering a broad evaluation context for the LLM-based tools.

3.2.3 Selecting repositories. We adopted a judgment sampling strategy to select public JavaScript projects on GitHub, applying strict inclusion criteria. Projects had to use JavaScript as the primary language (≥75% of the codebase), be open source, include a test suite, and use the Jest framework—chosen for its popularity, as confirmed by a late-2024 developer survey [14]. Only actively maintained repositories (last commit ≥ Jan 1, 2024) with ≥5,000 GitHub stars were considered.

Table 1: Technical Characteristics of Selected Projects

Repository	Stars	JS Usage(%)	Type
atlassian/react-beautiful-dnd	33.9k	99.99%	library
brookhong/surfingkeys	5.7k	75.35%	browser-extension
chrisleekr/binance-trading-bot	5.3k	98.54%	crypto trading bot
CodeGenieApp/serverless-express	5.2k	99.75%	framework
jackocnr/intl-tel-input	7.9k	94.99%	plugin
katex/katex	19k	85.82%	library
miragejs/miragejs	5.5k	95.71%	framework
prettier/prettier	50.4k	82.68%	formatter
shipshapecode/tether	8.5k	91.65%	library
verlok/vanilla-lazyload	7.8k	83.64%	library

Initial candidates were identified using the GitHub Search (GHS) dataset [9], filtering by language, recency, and popularity, yielding 774 repositories. A custom Python script[1] applied filters—code composition, language percentage, license, test suite, and Jest usage—reducing the set to 93. From these, we randomly selected 10 repositories, listed in Table 1.

3.2.4 LLM-based tools used. This study utilized two LLM-powered tools to assist in the refactoring of test smells: GitHub Copilot Chat (GPT-4o) [13] and Amazon CodeWhisperer [35]. Both tools were integrated into the VSCode[2] environment and used through inline interactions to support smell-specific refactorings. These tools were chosen for their seamless integration with modern development environments and their ability to interpret natural language instructions in conjunction with source code—key capabilities for effective LLM-assisted refactoring.

[1]The artifacts used in this study are in our replication package [https://doi.org/10.5281/zenodo.15997271]. Repository data was fetched via the GitHub API https://api.github.com, including language usage based on reported byte counts.
[2]https://code.visualstudio.com/

3.3 Refactoring Process

The second phase of the study focused on a structured and systematic refactoring process. We began by executing the original test suites and collecting code coverage data to establish baseline metrics for each repository. This ensured that subsequent modifications could be validated against the projects' original behavior.

To support this, we extended `SNUTS.JS` to analyze individual test files and developed a script to convert `Steel`'s output into a structured CSV format for integration[1]. With smells identified, we selected at least five instances of each type per repository, following predefined criteria to ensure a diverse yet manageable dataset. In total, 148 test methods were analyzed.

Refactorings were performed using zero-shot prompting, based on structured templates informed by prior literature [8, 18, 41] and pilot tests. Each prompt included metadata such as the smell category, its location, a descriptive explanation, and a request focused on preserving behavior and improving readability. An example is shown in Listing 1.

Prompt templates accounted for variations in how smell locations were reported: SNUTS.JS provided line ranges, while Steel yielded more granular line, column, and character data. This flexibility enabled uniform formatting while preserving location precision.

The process involved manual interaction guided by LLMs. Structured prompts were issued via GitHub Copilot Chat (GPT-4o) and Amazon CodeWhisperer in VSCode, simulating developer workflow. Although manually triggered to allow inspection, future automation could enhance reproducibility.

Listing 1: Prompt template

```
Context:I'm refactoring test smells from a test file to improve
code quality.
Issue Details:
Smell Category: {Test smell category}
Smell Location:
- Line Range: startLine: {startLine}, endLine: {endLine}
  or
- Line: {line}, Column: {column}, Index: {index}
Description:
{Detailed test smell description}
Request:
Refactor the affected code to eliminate the {Test smell category}.
Ensure the test remains correct, readable, and maintainable.
```

3.4 Post-refactoring Analysis

In the final phase of the study, we conducted a comprehensive analysis to evaluate the outcomes of the refactoring process. This phase aimed to validate the correctness of the refactored tests and assess whether the test smells were effectively mitigated. To begin, we re-executed the test suites for each project to ensure that the refactored tests maintained their intended behavior. This step was crucial for verifying that the refactoring did not introduce regressions or alter the semantic integrity of the tests. In parallel, we re-measured code coverage to detect any unintended changes in test coverage levels caused by the transformations.

Next, we reapplied the SNUTS.JS and Steel tools to the refactored code to determine whether the test smells previously identified had been successfully eliminated. By comparing the smell detection results before and after refactoring, we were able to quantify the effectiveness of each LLM-assisted transformation.

To deepen the analysis, we extracted software quality metrics from both original and refactored versions using a custom JavaScript script. This script leverages Babel[3] to parse code into Abstract Syntax Trees (ASTs), enabling automated and consistent metric collection[1]. The AST-based analysis computed six structural metrics: Logical SLOC (code volume), Cyclomatic Complexity (independent paths), Cyclomatic Density (complexity relative to size), Halstead Effort (cognitive load), Halstead Bugs (error prediction), and Maintainability Index (long-term maintainability).

By integrating functional verification, test smell detection, and structural metrics, our multi-dimensional evaluation assessed not only behavioral preservation but also improvements in code quality and maintainability from LLM-generated refactorings.

4 Results and Discussion

This systematic study investigated the effects of refactoring on automated tests using a sample of 148 identified cases across ten distinct software projects. The case selection focused on seven specific test smell categories with the following distribution: 43 instances of *Duplicate Assert*, 38 of *Magic Number*, 23 of *Lazy Test*, 22 of *Suboptimal Assert*, 12 of *Conditional Test Logic*, 7 of *Overcommented Test*, 2 of *Assertion Roulette*, and 1 of *Test Without Description*. The absence of two smell categories, *Sensitive Equality*, and *Redundant Print*, in the analyzed projects may reflect either their lower prevalence in real-world development contexts or project teams' adoption of preventive coding practices. The missing *Suboptimal Assert* cases particularly suggest developers may favor more expressive assertions or that modern testing tools inherently discourage such problematic patterns.

We defined an evaluation framework employing four points to assess the prompt-based refactoring impact: (i) test results changed indicating modifications in test outcomes, (ii) coverage change measuring test coverage variations, (iii) added new smell detecting newly introduced smells through SNUTS.JS and Steel analysis, and (iv) removed smell tracking original smell elimination. This multi-dimensional approach enabled the simultaneous evaluation of both functional test behavior and structural code quality improvements resulting from the refactoring interventions.

4.1 RQ1. Test Behavior and Coverage

To assess the functional correctness of LLM-assisted refactorings, we analyzed whether they yielded unchanged test results and coverage—indicating a likely preservation of behavior. Specifically, we examined changes in test execution outcomes (failures, deletions, or additions) and variations in coverage metrics across all 148 refactored test methods.

Test Behavior. Copilot altered test behavior in 22 cases: 17 led to new failures or removed tests, and 5 added new passing tests without removing the original smell. Failures were mostly linked to *Duplicate Assert* (7), *Conditional Test Logic* (4), *Suboptimal Assert* (4), and *Lazy Test* (2); all additions involved *Lazy Test*. **Whisperer** showed a similar pattern, with 21 behavioral changes—17 failures and 4 non-failure additions. Again, all non-failure additions

[3] https://babeljs.io/

involved *Lazy Test*. The distribution of smell types among the 17 failure cases was more diverse than Copilot: *Duplicate Assert* (6), *Lazy Test* (4), *Conditional Test Logic* (3), *Overcommented Test* (2), *Suboptimal Assert* (1), and *Assertion Roulette* (1).

Test Coverage. Copilot modified coverage in only 5 samples—two *Duplicate Assert* (coverage reduced) and three *Lazy Test* (coverage increased, smell remained). **Whisperer** impacted coverage in 6 samples, spanning *Duplicate Assert* (2), *Lazy Test* (2), and *Conditional Test Logic* (2), with mixed outcomes. Notably, complete smell removal for *Duplicate Assert* consistently coincided with reduced coverage, while for *Lazy Test* one sample showed increased coverage and the other decreased, with no successful removals. Regarding *Conditional Test Logic*, coverage increased in one sample and decreased in the other, with complete smell elimination.

Cross-tool insights. Among overlapping cases, *Lazy Test* dominated behavior modifications (54.5%) but caused failures in only 33% of them. In contrast, *Duplicate Assert* and *Conditional Test Logic*, though less frequent, led to failures in all affected samples, suggesting higher structural fragility. For coverage, both LLMs reduced it when removing *Duplicate Assert*, while yielding opposing outcomes on the same *Lazy Test* sample—Copilot increased coverage, Whisperer decreased.

Implications. Refactoring strategies must consider both the frequency and inherent risk of test smells. Smells like *Duplicate Assert* and *Conditional Test Logic* are more prone to breaking test behavior and reducing coverage, requiring cautious or guided interventions. Conversely, *Lazy Test*, while frequent, had more benign and inconsistent effects. Importantly, higher test coverage does not always indicate meaningful improvement, especially when smells remain. Divergent LLMs behaviors further emphasize the need for multi-tool assessments and smell-aware LLM prompts that balance structural impact with functional correctness.

4.2 RQ2. Introduction of New Test Smells

To assess whether the refactoring process introduced new test smells, we conducted a post-refactoring analysis using the same detection tools applied before transformation.

Copilot. Among the 148 refactored methods, Copilot introduced new smells in 19 cases. The majority (17) involved the *Lazy Test* smell, while the remaining 2 were new instances of *Duplicate Assert*.

CodeWhisperer. Whisperer introduced new smells in 24 cases, including 20 *Lazy Test*, 3 *Duplicate Assert*, and 1 *Conditional Test Logic* instance.

Cross-tool insights. *Lazy Test* was the only smell consistently introduced across both tools, appearing in 16 overlapping cases. In several of these, it co-occurred with other smells. Specifically, Copilot yielded 6 cases of *Lazy Test* compounded with smells such as *Conditional Test Logic*, *Eager Test*, or *Duplicate Assert*. Whisperer showed a similar pattern in 5 instances. Table 2 shows the new smells introduction to each smell type.

Implications. The results reveal a recurring challenge in LLM-assisted refactoring: the inadvertent creation of new or combined smells, especially involving Lazy Test. We plan to investigate these smell combinations in more depth to better understand their root causes. Future approaches should adopt smell-aware prompting and validation mechanisms that minimize the risk of introducing new issues during refactoring.

Table 2: Added Smells by Type

Smells	C	W	Smells added by type
Lazy Test	17	20	Lazy Test, Assertion Roulette, Eager Test, Duplicate Assert, Verbose Statement, Conditional Test Logic, Unknown Test, Non Functional Statement
Duplicate Assert	2	3	Duplicate Assert, Magic Number
Conditional Test Logic	0	1	Conditional Test Logic

4.3 RQ3. Removal Effectiveness by Smell Type

We evaluated the effectiveness of LLM-assisted refactoring by comparing the number of test smells removed and the number of successful removals, i.e., cases where the smell was eliminated without affecting test behavior or coverage. Table 3 presents the results, where *Rem.* refers to the total number of removed instances and *Succ.* denotes those considered successful. For example, among 22 instances of *Suboptimal Assert*, Copilot removed all but succeeded in only 18. In contrast, for *Magic Number*, it removed 27 out of 38, all without introducing regressions.

Overall performance. Copilot achieved a total removal rate of 69.59%, with 58.78% considered successful. Whisperer demonstrated slightly lower performance, with 54.05% total and 47.30% successful removal. This 15 percentage point gap suggests that the chosen refactoring strategy significantly influences outcomes.

Per-smell analysis. Both LLMs failed to remove the *Lazy Test* smell in all 23 cases. Their refactorings sometimes retained or reintroduced the smell, as shown in Table 2. In contrast, high success rates were observed for *Suboptimal Assert* (81.82% for Copilot; 95.45% for Whisperer) and *Magic Number* (71.05% for both tools). Notably, 62 refactored cases were successful across both tools, with *Magic Number* (28 cases) and *Suboptimal Assert* (22) as the most commonly and reliably addressed smells. Additionally, the presence of unsuccessful removals (10.81% for Copilot and 6.76% for Whisperer), where smell elimination affected test behavior, highlights the need for more sophisticated approaches that preserve original functionality.

Implications. These findings highlight the need for smell-specific strategies in automated refactoring. Smells with clear syntactic signatures, such as *Magic Number*, are well-suited to current LLM capabilities. In contrast, context-sensitive smells like *Lazy Test* require more advanced reasoning or prompt engineering. To enhance reliability, LLM-based tools could integrate validation steps and smell-aware logic that considers structural and behavioral complexity during refactoring. As next step, we plan a controlled experiment with professional developers to assess whether the removals classified as successful are indeed correct from a human perspective.

4.4 RQ4. Impact on Structural Code Quality

To assess how refactoring affected code quality, we compared six software metrics before and after transformation: Logical SLOC, Cyclomatic Complexity, Cyclomatic Density, Halstead Effort, Halstead Bugs Estimate, and Maintainability Index. Table 4 summarizes the results, reflecting the magnitude and direction of changes introduced by each tool.

Table 3: Results by Smell for Copilot (C) and Whisperer (W)

Smell Type	Rem. (C)	Succ. (C)	Rem. (W)	Succ. (W)
Assertion Roulette (2)	2	2	2	1
Conditional Test Logic (12)	9	6	6	4
Duplicate Assert (43)	36	27	18	12
Lazy Test (23)	0	0	0	0
Magic Number (38)	27	27	27	27
Overcommented Test (7)	6	6	5	4
Suboptimal Assert (22)	22	18	21	21
Test Without Description (1)	1	1	1	1

Table 4: Summary Statistics Before and After Refactoring

Metric	State	Mean	Median	Std. Dev.	IQR	Skewness
SLOC Logical	Before	12.662	10.500	10.004	9.250	1.928
	Copilot	13.534	11.000	9.050	10.000	1.557
	Whisperer	14.338	12.000	10.321	10.000	1.934
Cyclomatic	Before	1.622	1.000	1.505	0.000	2.708
	Copilot	1.615	1.000	1.593	0.000	2.771
	Whisperer	1.615	1.000	1.696	0.000	3.063
Cyclomatic Density	Before	15.987	14.835	9.079	16.667	0.511
	Copilot	13.612	11.111	7.754	12.308	0.824
	Whisperer	12.561	11.111	6.828	8.974	0.894
Halstead Effort	Before	1559.815	489.223	3091.044	984.966	3.116
	Copilot	1258.596	539.111	2257.025	867.238	3.995
	Whisperer	1578.095	629.366	2965.288	1017.767	3.430
Halstead Bugs	Before	0.017	0.013	0.012	0.013	1.548
	Copilot	0.017	0.015	0.011	0.011	1.461
	Whisperer	0.018	0.014	0.012	0.011	1.538
Maintainability	Before	95.739	100.000	8.137	5.806	−2.203
	Copilot	95.551	100.000	7.639	5.352	−1.941
	Whisperer	95.023	100.000	8.550	6.963	−1.940

Code size and complexity. Both tools increased Logical SLOC, with Whisperer showing greater expansion and variability. Copilot's median rose slightly (10.5 to 11.0) with a narrower interquartile range, while Whisperer reached a higher maximum (36 lines) and higher post-refactoring deviation. Cyclomatic Complexity remained stable for both, preserving control flow structure. However, Cyclomatic Density improved, decreasing by 2.4 points for Copilot and 3.4 for Whisperer, indicating better distribution of complexity relative to code size.

Cognitive effort and error-proneness. Copilot significantly reduced Halstead Effort (by 301 points), suggesting lower cognitive load. In contrast, Whisperer slightly increased it. Despite this, both tools maintained stable Halstead Bugs estimates, with nearly identical medians and interquartile ranges, indicating no change in theoretical error proneness.

Maintainability. Both tools preserved high maintainability scores (mean > 95, median = 100). Copilot yielded more consistent outcomes, with a smaller drop in the mean and tighter post-refactoring distribution. Whisperer exhibited a larger mean reduction and wider interquartile range, indicating more variability in maintainability impact.

Implications. The metrics reveal three key insights: (1) Both tools preserve core quality attributes, with minimal impact on complexity and maintainability. (2) Copilot adopts a conservative approach, reducing cognitive effort while keeping distributions tight. Whisperer applies a more aggressive strategy, achieving better structural optimization—evidenced by density gains—but at the cost of slightly increased code size and effort. (3) Despite different philosophies, both tools maintain high maintainability, highlighting complementary strengths in AI-assisted refactoring.

5 Threats to Validity

We discuss threats to the validity of this study Wohlin et al. [40] in terms of internal, external, and construct validity. Regarding **internal validity**, a threat lies in the use of only two LLMs—GitHub Copilot and Amazon CodeWhisperer—for refactoring test smells; however, these were chosen due to their widespread use in code analysis. Additionally, since refactoring by LLMs may alter test behavior, we validated the accuracy of the refactored tests and verified whether the test smells were effectively removed. In terms of **external validity**, the focus on JavaScript limits generalizability to other dynamically typed languages (e.g., Python). Although we analyzed only 10 projects, they were carefully selected real-world JavaScript systems, and the inclusion of 10 distinct test smells helps ensure a representative evaluation of JavaScript test code refactoring. Finally, concerning **construct validity**, a limitation is that zero-shot prompts may not handle all smell types effectively, and LLMs may even introduce new test smells. To mitigate this, we rigorously assessed refactoring quality by re-analyzing the outputs using SNUTS.JS and Steel tools.

6 Conclusions

This study investigated the use of Large Language Models (LLMs) to automatically refactor test smells in JavaScript. Evaluating 148 smell instances across ten projects, we found that both GitHub Copilot Chat and Amazon CodeWhisperer can effectively remove smells while preserving test behavior and coverage. Copilot achieved higher success rates, while Whisperer showed stronger performance on specific smell types. However, both tools also introduced new smells, especially Lazy Test. These results offer a first step toward integrating LLMs into smell-aware testing workflows, with promising but cautious potential. As future work, we plan to conduct a controlled experiment with professional developers to assess the human-perceived quality of the refactorings. We also intend to include other prompt strategies and perform statistical analysis to validate and deepen our current findings.

ARTIFACT AVAILABILITY

The artifacts used in this study, including the analyzed repositories, refactoring scripts, prompts, and complete results, are available at: https://doi.org/10.5281/zenodo.17058709.

ACKNOWLEDGMENTS

This work was partially supported by FAPESB (2025), UEFS-AUXPPG (2025), CAPES-PROAP (2025), CAPES, CNPq (Grant No. 312920/2021-0), CNPq (Grant No. 403361/2023-0) and FAPEMIG (Grant No. APQ-01488-24).

REFERENCES

[1] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In *Proceedings of the 25th Int. Conf. on Evaluation and Assessment in Software Engineering*. 170–180.

[2] Yasaman Amannejad, Vahid Garousi, Rob Irving, and Zahra Sahaf. 2014. A Search-Based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. 302–311.

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 56–65.

[4] Manuel Breugelmans and Bart Van Rompaey. 2008. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 11.

[5] I. Burnstein, T. Suwanassart, and R. Carlson. 1996. Developing a Testing Maturity Model for software test process evaluation and improvement. In *Proceedings International Test Conference 1996. Test and Design Validity*. 581–589.

[6] Stephen Cass. 2024. The Top Programming Languages 2024. *IEEE Spectrum* (August 2024). https://spectrum.ieee.org/top-programming-languages-2024 Accessed: 2025-04-28.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2024. An Empirical Study on the Code Refactoring Capability of Large Language Models. arXiv:2411.02320 [cs.SE] https://arxiv.org/abs/2411.02320

[9] 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)*. 560–564.

[10] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. SoCRATES: Scala radar for test smells. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. Association for Computing Machinery, 22–26.

[11] Sakina Fatima, Hadi Hemmati, and Lionel Briand. 2024. Flakyfix: Using large language models for predicting flaky test fix categories and test code repair. *IEEE Transactions on Software Engineering* (2024).

[12] Yi Gao, Xing Hu, Xiaohu Yang, and Xin Xia. 2025. Context-Enhanced LLM-Based Framework for Automatic Test Refactoring. In *International Conference on the Foundations of Software Engineering (FSE 2025)*.

[13] GitHub. 2024. About GitHub Copilot Individual. https://docs.github.com Accessed: 2024-09-29.

[14] Sacha Greif and Eric Burel. 2024. The State of JavaScript 2024: Testing - Jest. https://2024.stateofjs.com/en-US/libraries/testing/. Online; accessed 25 April 2025. Survey run by Devographics from Nov 13 to Dec 10, 2024 with 14,015 responses. Results published on Dec 16, 2024..

[15] Navid Bin Hasan, Md Ashraful Islam, Junaed Younus Khan, Sanjida Senjik, and Anindya Iqbal. 2025. Automatic High-Level Test Case Generation using Large Language Models. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society.

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

[17] Dalton Jorge, Patricia Machado, and Wilkerson Andrade. 2021. Investigating Test Smells in JavaScript Test Code. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*. Association for Computing Machinery, 36–45.

[18] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. 22199–22213.

[19] Negar Koochakzadeh and Vahid Garousi. 2010. Tecrevis: a tool for test coverage and test redundancy visualization. In *International Academic and Industrial Conference on Practice and Research Techniques*. Springer, 129–136.

[20] Keila Lucas, Rohit Gheyi, Elvys Soares, Márcio Ribeiro, and Ivan Machado. 2024. Evaluating large language models in detecting test smells. *arXiv preprint arXiv:2407.19261* (2024).

[21] Rian Melo, Pedro Simões, Rohit Gheyi, Marcelo d'Amorim, Márcio Ribeiro, Gustavo Soares, Eduardo Almeida, and Elvys Soares. 2025. Agentic SLMs: Hunting Down Test Smells. arXiv:2504.07277 [cs.SE] https://arxiv.org/abs/2504.07277

[22] Gerard Meszaros, Shaun M. Smith, and Jennitta Andrea. [n. d.]. The Test Automation Manifesto. In *Extreme Programming and Agile Methods - XP/Agile Universe 2003* (2003). Springer Berlin Heidelberg, 73–81.

[23] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 1–5.

[24] Henriquer Nunes, Eduardo Figueiredo, Larissa Soares, Sarah Nadi, Fischer Ferreira, and Geanderson Esteves. 2025. Evaluating the Effectiveness of LLMs in Fixing Maintainability Issues in Real-World Projects. In *32th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[25] Jhonatan Oliveira, Luigi Mateus, Tássio Virgínio, and Larissa Rocha. 2024. SNUTS.js: Sniffing Nasty Unit Test Smells in Javascript. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*. SBC, Porto Alegre, RS, Brasil, 720–726.

[26] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the diffusion of test smells in automatically generated test code: an empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*. Association for Computing Machinery, 5–14.

[27] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1650–1654.

[28] Adriano Pizzini. 2022. Behavior-based test smells refactoring: toward an automatic approach to refactoring eager test and lazy test smells. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. Association for Computing Machinery, New York, NY, USA, 261–263.

[29] Adriano Pizzini, Sheila Reinehr, and Andreia Malucelli. 2023. Sentinel: A process for automatic removing of Test Smells. In *Proceedings of the XXII Brazilian Symposium on Software Quality*. Association for Computing Machinery, New York, NY, USA, 80–89.

[30] Rudolf Ramler and Klaus Wolfmaier. 2006. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 International Workshop on Automation of Software Test*. Association for Computing Machinery, 85–91.

[31] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. 2007. Rule-based Assessment of Test Quality. *J. Object Technol.* 6, 9 (2007), 231–251.

[32] Railana Santana, Luana Martins, Tássio Virgínio, Larissa Rocha, Heitor Costa, and Ivan Machado. 2024. An empirical evaluation of RAIDE: A semi-automated approach for test smells detection and refactoring. *Science of Computer Programming* 231 (2024), 103013.

[33] Andrew Costa Silva. 2022. Identificação e Caracterização de Test Smells em JavaScript. *Instituto de Ciencias Exatas e Informática - Pontifícia Universidade* 138 (2022), 52–81.

[34] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. 2023. Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date? *IEEE Transactions on Software Engineering* 49, 3 (2023), 1152–1170.

[35] A. Team. 2023. Copilotchat. https://docs.aws.amazon.com/codewhisperer/ Accessed: 2025-04-25.

[36] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 4–15.

[37] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. JNose: Java Test Smell Detector. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. Association for Computing Machinery, 564–569.

[38] Tássio Virgínio, Luana Martins, Railana Santana, Adriana Cruz, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. On the test smells detection: an empirical study on the JNose Test accuracy. *Journal of Software Engineering Research and Development* 9, 1 (Sep. 2021), 8:1 – 8:14.

[39] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2021. PyNose: A Test Smell Detector For Python. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 593–605.

[40] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[41] Tingyu Xie, Qi Li, Jian Zhang, Yan Zhang, Zuozhu Liu, and Hongwei Wang. 2023. Empirical Study of Zero-Shot NER with ChatGPT. arXiv:2310.10035 [cs.CL] https://arxiv.org/abs/2310.10035

[42] Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. 2023. LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. 206–217.

[43] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot's Impact on Productivity. *Commun. ACM* 67, 3 (2024), 54–63.