Code Smells in React Projects of a Junior Enterprise

Victor Peixoto Barbosa
Computer Science Department, UFBA
Salvador, Brazil
victorrpeixotoo@gmail.com

Cláudio Sant'Anna
Computer Science Department, UFBA
Salvador, Brazil
claudionsa@ufba.br

Abstract

This study investigates the prevalence and correlation of code smells in web projects developed by a Junior Enterprise (JE) using the React or Next.js framework. Through a quantitative analysis of 40 projects using the ReactSniffer2 tool and custom scripts, 14 distinct code smells were evaluated. The results reveal a high density of smells, averaging 19.4 occurrences per 1,000 lines of code (LOC). String Literals was identified as the most prevalent smell, occurring 4.75 times more frequently than Mutable Variables. Furthermore, the study identified eight strong correlations between smells, all of which reached statistical significance ($p < 0.05$). These findings provide empirical evidence to support the enhancement of developer training and curricula in student-led organizations, aiming to improve software maintainability and code quality early in these students' careers.

Keywords

Code smells, React, Junior enterprise, Static analysis, Software engineering education.

1 Introduction

The landscape of web development has been transformed by versatile frameworks and libraries. According to recent industry surveys in 2025 [1], JavaScript remains the most popular language, with React and Next.js leading the front-end ecosystem. The dominance of these technologies is driven by their flexibility and the high market demand for developers capable of navigating their complex ecosystems, which often include integrated tools like Tailwind CSS and Bootstrap. Consequently, a significant number of undergraduate students enter the professional market through web development centered on these frameworks. They often find themselves overwhelmed by the variety and complexity of each flavor these technologies provide to the base web development routine.

In Brazil, the Junior Enterprise (JE) movement provides a unique environment in which students establish and manage their own companies while still at university. This initiative offers undergraduates their first professional experience in a real-world setting. Within technology-focused JEs, React and Next.js are the standard stacks for web projects due to their popularity. As enrollment in technology courses continues to increase globally [2], understanding how these students transition to the market and ensuring that they adopt best practices is critical to their long-term professional success.

The rapid pace of learning in student-led environments can lead to the introduction of code smells [5]. Code smells are symptoms of poor design or implementation choices that, while not necessarily bugs, hinder maintainability and increase technical debt. While existing literature explores code smells in various contexts [4], there is a lack of empirical research focusing specifically on software developed by undergraduate volunteers in JEs. This demographic is particularly vulnerable to "learning traps" during the adoption of modern frameworks like React and Next.js as they enter a fast-paced industry where fundamental concepts are sometimes overlooked in favor of achieving a rapid, yet potentially superficial, grasp of frameworks and their ecosystem of tools.

This work analyzes projects developed by students in a Junior Enterprise to investigate the prevalence and correlation of code smells. By identifying these patterns, we aim to provide data that can inform training programs in similar organizations, mitigating the introduction of harmful practices early in a developer's career. To guide this investigation, we address two research questions:

RQ1: Which code smells are most prevalent in web projects developed by a Junior Enterprise?

RQ2: What are the significant correlations between these prevalent code smells?

By answering these questions, we hope to gain a deeper understanding of the learning curve of young developers. Our goal is to highlight critical areas where attention is needed to prevent long-term maintenance issues and support the growth of high-quality professionals.

The remainder of this paper is organized as follows. Section 2 presents work related to ours. The study methodology is described in Section 3. Section 4 presents and discusses the results. Section 5 discusses threats to validity. Finally, Section 6 presents conclusions and future work.

2 Related Work

Code smells are widely established concepts in software engineering, and their recognition is vital for developing cleaner, more maintainable code [5]. In recent years, empirical studies on code smell prevalence have expanded into specialized niches, particularly within the web development ecosystem due to the dominance of frameworks like React.

A foundational study by Ferreira and Valente [4] explored the specific dynamics of the React ecosystem. The authors proposed a novel set of React-specific code smells and introduced *ReactSniffer*, a tool designed for their static detection. They evaluated the historical evolution of the ten most popular React projects on GitHub, tracking how frequently these smells were refactored. While their work established the technical baseline for React static analysis, our study shifts the focus to a different subset of projects: those developed by undergraduate student volunteers within a Junior Enterprise (JE) environment.

The investigation of code smell prevalence spans several other specialized domains. For instance, Muse et al. [9] conducted a large-scale study on the prevalence and evolution of SQL-specific code smells across 150 projects. Their findings revealed that smells predominantly emerge during the earliest stages of a project's lifecycle

and are rarely refactored later—a phenomenon that heavily resonates with the rapid development cycles observed in student-led organizations.

Beyond structural maintainability, bad practices have also been explored under the lens of security. Gadient et al. [6] investigated *security smells* in thousands of Android mobile applications, discovering that up to 69% of the analyzed apps suffered from at least three distinct types of security smells, highlighting how organizational context and complexity dictate the introduction of poor practices. Furthermore, Mahbub et al. [8] extended this analysis to simulation modeling software, comparing 155 simulation systems against 327 traditional GitHub repositories. They identified a high incidence of domain-specific smells, such as *Long Statement* and *Magic Number*, reinforcing that every distinct technology stack and development environment fosters its own dominant architectural and implementation issues.

This plurality of research underscores that while code smells are universal, each framework and programming paradigm introduces unique patterns and thresholds. By leveraging *ReactSniffer2*—an updated fork of the original tool proposed in [4]—this work fills a major gap in the literature by examining the prevalence, correlation, and educational implications of code smells within the specific, high-turnover context of student-run software enterprises.

3 Methodology

Figure 1 shows the workflow of our study methodology and the following subsections describe each step.

3.1 Project Selection

We selected the projects from a junior enterprise of which one of the authors of this work used to be a member and which allowed the use of its projects for analysis in this work. This brings the uniqueness of the company being composed entirely of undergraduate students, providing a distinctive perspective to the field.

The primary motivation for this work was analyzing code smell prevalence on the projects. To this end, we selected 40 projects from the junior enterprise. All the projects were developed by teams of unpaid students. Despite originating from the same company, due to the span of years in which they were developed and the high turnover rate inherent to JEs nature, there is minimal overlap between members of each project. This allowed a vast number of different developers to be involved in the projects analyzed in this study.

Given the nature of the research, only projects using *React* or *Next.js* were considered. Next.js is a framework built on top of *React*. This study benefits from utilizing tools and smell characteristics appropriate for this environment, allowing a look into how people entering the industry may use poor practices involving various *React* patterns. This is valuable due to the sheer relevance and market dominance of *React*.

The projects were divided into four categories: Websites, Systems, Internal, and External. Websites and Systems are mutually exclusive, as are Internal and External. Therefore, each project is designated as either a Website or a System and, additionally, as either Internal or External.

Definitions for Website and System projects are present in the company's project proposal documents: Systems are projects that include back-end aspects and database integration, while Websites do not involve this level of integration, being, at most, managed by a Headless CMS (Content-Management-System) to assist in content management without the need for a custom back-end. Regarding Internal and External definitions, Internal projects are those where the company itself is the client, while External projects involve a third party.

Among the 40 projects, 10 are Systems and 30 are Websites. Similarly, 30 are External projects, while the other 10 are Internal. We analyze the results in terms of these categories in Section 4.3.

3.2 Codebase Preparation

The company granted access to its GitHub repository. The preparation consisted of cloning each project into the same directory using the `git clone` command.

A project folder was preserved in the directory if it included a front-end and utilized *React* or *Next.js*, otherwise, the project was excluded. The 40 remaining projects formed the directory used for the following steps.

3.3 Data Extraction

For data extraction, we used the ReactSniffer2 tool [7], which is an evolution of the ReactSniffer tool [3]. We preferred ReactSniffer2 because it implements the detection of additional smells, outputs data in JSON format and is compatible with more recent versions of *React*. The original tool, ReactSniffer, outputs data in two different EXCEL spreadsheets for each execution, while ReactSniffer2 generates everything in a single JSON file, facilitating further processing. Additionally, ReactSniffer failed to identify a high number of source code files in most of the 40 projects. This occurred presumably because it does not account elements that have recently appeared or been changed in the *React* environment since its release. ReactSniffer2 successfully identified all files in every project. Notably, the original ReactSniffer had its last commit on July 1, 2022, while ReactSniffer2 was last updated on February 29, 2024.

ReactSniffer2 can identify up to 16 code smells. Table 1 lists all the 16 code smells alongside with a brief explanation for each. Additionally, a folder can be found in our repository[1] with 16 files exemplifying each of the smells in further detail.

When applied to a project, ReactSniffer2 performs a static analysis producing a *.JSON* file, which lists the components of the project identifying the code smells presented in each component. Also it computes the number of Lines of Code (LOC) per source code file that implements each component. We used this *.JSON* file to build a spreadsheet for computing statistics.

During the execution of the study, we ran the tool for each project. Since the tool always creates a *.JSON* with the same name, we renamed the file to smells-[project-name] before proceeding to the next project. At the end of this stage, *.JSON* files for all projects were ready in the same directory.

[1] https://doi.org/10.5281/zenodo.21853870

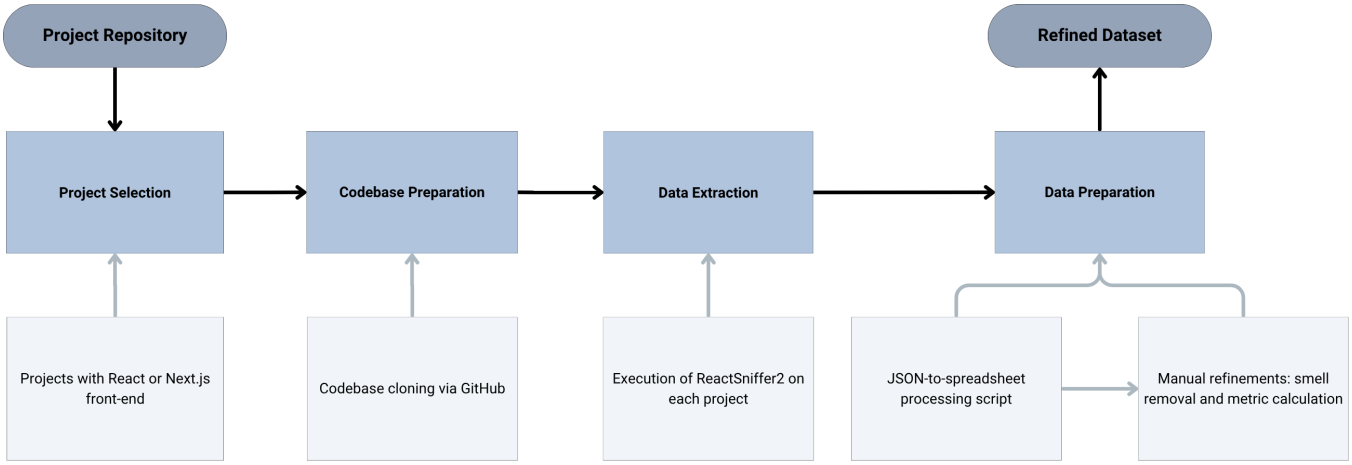

Figure 1: Methodological workflow detailing the extraction, processing, and statistical preparation of the React project dataset.

Table 1: Catalog of investigated React code smells and brief descriptions.

Smell Name	Brief Explanation
Props in Initial State (PIS)	Copying a prop value into local state during initialization, causing synchronization bugs if the prop updates later.
Use of Index as Key (AIK)	Using array loop indexes as element keys, forcing unnecessary DOM recreations and corrupting complex internal component states.
Component Nesting (JSX)	Defining a child component inside a parent component, causing it to unmount and remount on every parent render cycle.
Large Component (LC)	Monolithic components consolidating multiple business domains, side effects, form controls, and extensive layout trees.
Prop Drilling (PD)	Threading properties through multiple intermediate layout components that have no use for the data themselves.
Too Many useState (TMU)	Declaring scattered, individual hooks for tightly coupled state values instead of consolidating them into structural state objects.
Direct DOM Manipulation (DOM)	Modifying the layout or styles directly through imperative selection commands, completely bypassing React's Virtual DOM reconciliation.
Props Spreading (PS)	Indiscriminately forwarding an entire props collection down to native HTML wrappers, leading to invalid DOM attributes.
Deep Indentation (DI)	Writing heavily nested ternary operators or inline logical expressions within the visual layout section, lowering overall code legibility.
Too Many Props (TP)	Creating components with an oversized, uncohesive property signature that mixes presentation config, domain data, and event callbacks.
Large useEffect (LUE)	Packaging asynchronous fetching, page-meta mutations, and persistent listeners into a single side-effect hook with an oversized dependency array.
Mutable Variables (MV)	Modifying state instances directly in place, preventing React from identifying structural modifications due to shallow reference equality.
Procedural Patterns (PP)	Executing manual, imperative UI state commands inside interaction handlers instead of relying purely on a declarative, reactive architecture.
String Literals (SL)	Utilizing raw, hardcoded magic strings for conditionals and domain constraints rather than formal, immutable constants or contracts.
Use PrevState (PREVS)	Disregarding functional state updates when queueing values that depend on prior states, capturing a stale snapshot instead.
Never Using Class Components (CD)	Implementing functional components with hooks instead of class components.

3.4 Data Preparation

3.4.1 JSON → Spreadsheet. We developed a Python script to aggregate data from all *.JSON* files in the directory. This script is available in our open repository.

The script generates a spreadsheet (.xlsx) with two tabs:

(1) **Project Data**: A table with one line for each project containing the following columns: Project Name, Amount of LOC, Smell Density per LOC, Total Amount of Smells. In addition, the table contains one column for each of the 16 code smells with the number of each smell identified in each project.
(2) **Smell Analysis**: A table with one line for each code smell containing the following columns: Smell Name, Total Occurrences across projects, Mean per Project, Standard Deviation, Min, 25%, 50% (Median), 75%, Max, and arrays of project names for projects with the most and least occurrences.

The goal was to have a comprehensive, well-formatted, and standardized file for further statistical manipulation.

3.4.2 Spreadsheet Adjustments. Further changes were necessary for the purposes of this study, conducted via Google Spreadsheets due to its versatility.

For the sake of confidentiality, in the **Project Data** table, we added a column to identify projects by indices in the format "PXX" (P for project, XX from 01 to 40) and removed the project names. We also added two columns for identifying the Website/System and Internal/External categories of each project. We added a final column titled "Smell Density per 1,000 LOC", multiplying the "Smell Density per LOC" values by 1,000 for a more tangible understanding of the data. Finally, a "Total" row was added to sum the LOC and smell counts of all projects.

3.4.3 Data Refinements. Although detected by the tool, the "Never Using Class Components" smell (Table 1) is no longer considered as a bad practice in current day usage, since the standard has become not using class components at all, and utilizing hook components instead. Therefore, we removed the "Never Using Class Components" row from the **Smell Analysis** table.

Another alteration for further statistical analysis has been the removal of the smell "Large useEffect" due to the smell not being detected once in any of the 40 projects analyzed for this study. This smell is still significant and should be treated and accounted for, however, due to its absence, it has been omitted from calculations moving forward for this particular dataset. This also means that a total of 14 code smells are being processed in the statistical analysis of this study.

Lastly, a Spearman tab was created, containing tables for the Spearman correlation matrix of the smells, p-values for these correlations, and the correlation between LOC and total smells. All calculations were performed using standard Google Spreadsheets functions based on the cells generated by the script.

The spreadsheets and the Python script developed for data manipulation are available in our research artifact repository. To maintain organizational confidentiality, the provided datasets are anonymized versions of the internal project files, with names redacted or replaced by identifiers. This version remains fully sufficient for verification and replication purposes, as the project indices used throughout this paper correspond exactly to those in the publicly available spreadsheet.

Table 2: General statistics for detected smells.

Smell Name	Total	Mean	Mdn	Max	% Total	% Proj
SL	1007	25.17	10	274	43.38%	90%
MV	212	5.3	0	89	9.13%	45%
LC	199	4.97	3	33	8.57%	87.5%
PS	194	4.85	1	32	8.35%	67.5%
PIS	155	3.87	0	37	6.67%	47.5%
PD	118	2.95	1	22	5.08%	52.5%
AIK	111	2.77	1	16	4.78%	52.5%
PREVS	99	2.47	2	9	4.26%	75%
JSX	81	2.02	0	65	3.48%	25%
DOM	70	1.75	0	27	3.01%	45%
Others	52	-	-	-	2.24%	-

4 Results and Discussion

4.1 General Smell Perspective

Across 40 projects, we identified 2,321 smell instances. The average density was 19.4 smells per 1,000 lines of code (LOC), indicating that a smell occurs roughly every 50 lines. While Ferreira and Valente [4] analyzed historical data from the 10 most popular GitHub projects, our study focuses on the reality of Junior Enterprises (JE). For context, SonarSource technical guidance considers a rate of 10 or more smells per 1,000 LOC a "needs action" warning [11]. The high density found here could also suggest that student-led environments are particularly prone to these issues, aligning with Muse et al. [9], who observed that smells often emerge early in a project's lifecycle and are rarely adjusted later.

Table 2 shows that String Literals is the most common smell, taking place in 90% of the analyzed projects as well as the most prevalent smell (43.38%), occurring 4.75 times more than the second most common. This dominance may suggest that students may often neglect constants or internationalization during rapid development. Similar to Mahbub et al. [8], who found high incidences of specific smells in simulation software, our results show that each ecosystem has its dominant issues. Table 2 also shows that structural smells, such as Large Components and Props Spreading, appear frequently, possibly indicating a lack of component-based discipline.

4.2 Correlational Perspective

Due to the distribution metrics presented in Table 2, Spearman's Rank Correlation was selected over other options, such as Pearson, for the correlational needs of this study. Unlike Pearson, which assumes bivariate normality and is sensitive to extreme values, Spearman evaluates monotonic relationships by converting raw values into ordinal ranks. This rank-based approach mitigates the distortions caused by the skewed distributions and outliers, where it is seen for smells such as String Literals, which has a mean higher than its median and a max value almost 10 times higher than its mean. For interpreting Spearman's coefficient, we adopt

Table 3: Strongest code smell correlations identified ($r_s > 0.50$, all $p < 0.001$).

Smell A	Smell B	Spearman r_s	p-value
LC	SL	0.6505	< 0.0001
MV	PP	0.6057	< 0.0001
AIK	MV	0.5929	0.0001
PIS	SL	0.5824	0.0001
LC	MV	0.5797	0.0001
PIS	DOM	0.5634	0.0002
DI	PP	0.5349	0.0004
PIS	LC	0.5285	0.0005
PIS	MV	0.5110	0.0008

the guidelines from Newcastle University[10], where they have material dedicated to this correlation, categorizing correlations as moderate ($0.40 \leq |r_s| < 0.80$) or strong ($0.80 \leq |r_s| \leq 1.00$).

Out of 91 unique correlations, none were considered "strong" and 13 were considered "moderate", out of the moderate ones, all were positive correlations. Table 3 shows in more detail the 9 correlations that exceeded the 0.5 value for their coefficient including their p-values, which indicates just how unlikely the correlation is to have occurred by random chance. For this study purposes, the well accepted value of 0.05 was set as a threshold to consider significant correlations, i.e., any correlation with $p < 0.05$ is considered significant. All 13 moderate correlations found are considered significant $p < 0.05$.

Large Components and String Literals have the highest coefficient (r_s=0.65, $p < 0.001$), a possible speculation for this correlation is that, as components grow massive, hard-coded text tends to follow due to configurations or inline styles as JSX layout responsibilities grow. Mutable Variables and Procedural Patterns ranked second (r_s=0.60, $p < 0.001$), the speculation offered for this pairing is from a tendency of procedural programming habits as the members of the JE, who are also students, leak from their prior programming experience, which tends to involve procedural programming in early stages at universities. Lastly for this section, Use of Index as Key and Mutable Variables ranks third (r_s=0.59, $p < 0.001$), the hypothesis for this is a misunderstanding of React's reconciliation algorithm and component lifecycle states as imperative programming tendencies surface, considering the not proper utilization of React's immutable objects for lists and other components in need of the key attribute.

An insightful finding is that, when examining the correlation between LOC and Smell Density, it was only found a coefficient r_s=0.1955 with a p-value of $p = 0.2267$, showing that, with this particular dataset, there is no strong argument to be made that bigger projects have a higher smell density of code smells, only a weak and not statistically significant positive trend.

4.3 Project and Categorical Perspective

As noted in security smell studies [6], environment and complexity dictate the frequency of bad practices. Hence, in this section, we analyze the distribution of smell occurrences in terms of the Internal/External and Website/System categories. In the JE context, external clients bring tighter deadlines and higher pressure. Students, balancing academic and professional lives, often prioritize "getting the job done" over the "how it is done", falling into the "learning traps" mentioned in our introduction. Systems, being naturally more complex than Websites, aggravate these issues, acting as a fertile ground for the accumulation of smells. These situations are depicted in the findings shown in Table 4, where it is possible to see that there are more projects in the External or System categories with a high number of smell occurrences than in the Internal or Website categories, respectively.

5 Threats to Validity

This section identifies the potential threats to the validity of our findings and the measures taken to mitigate them.

5.1 Construct Validity

The primary threat concerns the definitions and thresholds used for code smell detection. As discussed, ReactSniffer2 uses thresholds that were partially established in previous scientific work [4] and later expanded by the community [7]. These thresholds might not perfectly align with every developer's perception of a "smell". The goal in looking for this updated version was to exactly mitigate the issues of outdated concepts and practices, given how quickly the framework and best practices have been updated.

5.2 Internal Validity

Our study is strictly quantitative, which limits our ability to establish definitive causality for the observed patterns. Due to time constraints and the availability of Junior Enterprise members, qualitative methods such as interviews or surveys were not feasible. Consequently, our discussions regarding the "reasons" behind the smells (such as deadline pressure or learning curves) are purely speculative. However, these speculations are informed by the authors' domain expertise, having spent several years embedded in the Junior Enterprise environment, providing a grounded perspective on the results, which is to say there is a robust foundation for the interpretations presented.

5.3 External Validity

Regarding generalizability, our dataset consists of 40 projects from a single Junior Enterprise in Brazil. While this represents a specific niche, the high turnover rate inherent to student-led organizations—where new management and developers cycle through every few semesters—means that these 40 projects involve a diverse range of developers and coding styles. This diversity provides a broader view of how different "generations" of students handle React development, though we acknowledge that a multi-institutional study would be required to generalize these findings to the entire Junior Enterprise movement.

6 Conclusion and Future Work

This study provided a quantitative analysis of code quality regarding code smells in React projects developed by undergraduate students in a Junior Enterprise (JE). By analyzing 40 projects, we identified patterns that reflect the specific challenges and learning

Table 4: Comparison by project scope and client.

Category	Number of Projects	Mean of Number of Smells per Project	SD	Min	25%	Median	75%	Max
Internal	10	42.1	65.94	3	7.75	23.5	38.75	225
External	30	63.3	70.30	3	20.25	37.5	84.25	299
Website	30	41.4	57.41	3	12.25	23.5	42.25	299
System	10	107.9	79.65	10	63.25	91	113.5	266

curves faced by developers at the beginning of their professional journeys.

6.1 Answering the Research Questions

RQ1: Which code smells are most prevalent in web projects developed by a Junior Enterprise?

The results indicate a clear dominance of specific bad practices: (i) **String Literals** (43.38%) was the most dominant smell, with a frequency 4.75 times higher than the runner-up, suggesting that "hard-coding" strings is a deeply established habit among this company's developers while appearing in 90% of the analyzed projects; (ii) **Mutable Variables** (9.13%) and **Large Components** (8.57%) followed, reflecting a lingering imperative programming mindset and a lack of experience with React's componentization potential, with Mutable Variables present in 45% of projects and Large Components present in 87.5%, displaying high spread across the dataset; and (iii) **Props Spreading** (8.35%) and **Props in Initial State** (6.67%) highlighted unfamiliarity with advanced React patterns (e.g., the `useContext` hook) and a misunderstanding of the "state vs. props" paradigm, appearing in 67.5% and 45.5% of analyzed projects, respectively.

RQ2: What are the significant correlations between these prevalent code smells?

Using Spearman's coefficient and filtering for statistical significance ($p < 0.05$), we identified key speculative synergies: (i) **Large Components and String Literals** ($r_s = 0.65, p < 0.001$), where overloaded components handling excessive layout responsibilities naturally accumulate inline text, configuration, and hard-coded values; (ii) **Procedural Patterns and Mutable Variables** ($r_s = 0.60, p < 0.001$), which reflects imperative programming habits leaking from early university coursework into declarative React development; and (iii) **Use of Index as Key and Mutable Variables** ($r_s = 0.59, p < 0.001$), pointing to local array mutation and a misunderstanding of Virtual DOM reconciliation, leading to unstable list identification.

These findings provide a baseline for Junior Enterprises to improve their developer onboarding and training programs by targeting the specific "learning traps" identified in this study. In total, 13 moderate correlations were identified as statistically significant ($p < 0.05$), with the full correlation matrix available in the project repository. Because this study does not include a qualitative evaluation to establish direct causation, the explanations offered throughout this work remain speculative. Nevertheless, they serve a dual purpose: (i) providing testable hypotheses for future empirical studies, and (ii) offering practical diagnostic heuristics for JEs to identify, evaluate, and mitigate code smell patterns within their own teams.

6.2 Future Work

To build upon these results, we suggest four primary research avenues: (i) **Qualitative Analysis,** conducting interviews or surveys with JE members to understand the "why" behind these smells, complementing this quantitative data with developer perspectives; (ii) **Causality and Education,** investigating the link between undergraduate CS curricula and the prevalence of imperative-style smells in React to adjust teaching methods; (iii) **Niche Diversification,** applying this methodology to students working in non-JE environments or across multiple Junior Enterprises to verify if these patterns are universal to the student demographic; and (iv) **Impact of Interventions,** developing and testing specific training modules in JEs aimed at mitigating the most prevalent smells and measuring their impact on code quality over time.

Artifact Availability

To support the reproducibility of this research, we provide a package of artifacts containing: (i) the Python script developed for processing the raw JSON data extracted by the ReactSniffer2 tool; (ii) the anonymized refined dataset in spreadsheet format, including the code smell frequencies, correlations, and statistical calculations for the 40 analyzed projects; and (iii) a folder with 16 examples for each of the code smells identified by ReactSniffer2 to more clearly explain each of them.

These items are intended to facilitate comparisons with related studies, enable replication in different contexts, or serve as a baseline for future research expansions. Due to non-disclosure agreements with the Junior Enterprise, the raw source code, repository names and extracted JSON from ReactSniffer2 cannot be made publicly available in the repository and have been redacted from the provided spreadsheet. This also means that the repository cannot contain a fully end-to-end replicative path due to lacking the project data that was used by the script that can be found there.

The artifacts are available at: https://doi.org/10.5281/zenodo.21853870

Acknowledgements

The authors would like to thank the Junior Enterprise involved in this study for providing access to the project repositories. For compliance with the VEM 2026 Generative AI policy, the authors acknowledge the use of Gemini (Google) to generate the accessibility descriptions (`\Description`) for the figures and to assist in the English grammatical refinement of the manuscript.

References

[1] 2025. *Stack Overflow Developer Survey 2025.* Retrieved May 2, 2026 from https://survey.stackoverflow.co/2025/technology

[2] Coursera. [n. d.]. *10 Most Popular College Majors.* Retrieved May 2, 2026 from https://www.coursera.org/articles/most-popular-college-majors
[3] Fabio Ferreira. 2022. *React-Code-Smells Repository.* Retrieved May 2, 2026 from https://github.com/fabiosferreira/React-Code-Smells
[4] Fabio Ferreira and Marco Valente. 2023. Detecting Code Smells in React-based Web Apps. *Information and Software Technology* 155 (March 2023), 1–35. doi:10.1016/j.infsof.2022.107111
[5] Martin Fowler and Kent Beck. 2018. *Refactoring.*
[6] Pascal Gadient, Marc-Andrea Tarnutzer, Oscar Nierstrasz, and Mohammad Ghafari. 2021. Security Smells Pervade Mobile App Servers. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21).* ACM, 1–10. doi:10.1145/3475716.3475780
[7] lsm 5. 2024. *ReactSniffer2 Repository.* Retrieved May 2, 2026 from https://github.com/lsm-5/reactsniffer2
[8] Riasat Mahbub, Mohammad Masudur Rahman, and Muhammad Ahsanul Habib. 2024. *On the Prevalence, Evolution, and Impact of Code Smells in Simulation Modelling Software.* arXiv:2409.03957 [cs.SE] https://arxiv.org/abs/2409.03957
[9] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. 2020. On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20).* ACM, New York, NY, USA, 327–338. doi:10.1145/3379597.3387467
[10] Newcastle University. [n. d.]. *Strength of Correlation.* Retrieved August 6, 2026 from https://www.mas.ncl.ac.uk/ask/numeracy-maths-statistics/statistics/regression-and-correlation/strength-of-correlation.html
[11] SonarSource Community. 2024. *What is the acceptable rate of code smells?* Retrieved August 6, 2026 from https://community.sonarsource.com/t/what-is-the-acceptable-rate-of-code-smells/131227