Analyzing Linter Usage and Warnings Through Mining Software Repositories: A Longitudinal Case Study of JavaScript Packages

Tjaša Heričko, Boštjan Šumak
Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia
{tjasa.hericko, bostjan.sumak}@um.si

***Abstract*—Linters are static analysis tools supporting developers by automatically detecting possible code errors, bad coding practices, violations of coding conventions, and styling issues while offering actionable advice on how to resolve these concerns to prevent inconsistencies in the codebase, errors, and degrading software quality. They are especially beneficial for software projects based on dynamically-typed languages, e.g., JavaScript, which are more prone to development errors and inconsistencies due to languages' dynamic nature. However, little is known about how their usage and generated warnings evolve throughout the entire development histories of real-world software projects. This paper performs source code-based commit history analysis on 15 JavaScript npm packages. The empirical analysis of 16,562 commits revealed that the choice of linter and their configurations are rarely subjected to change once introduced to a package. When configurations do change, most modifications are minor. Unresolved violations of rules, which produce warnings, generally increase. Though, the density of warnings was found to be slightly decreasing with the maturity of a package – overall and for each distinct group of warnings.**

Keywords*—*software quality assurance, static code analysis, lint, code repositories, npm packages, JavaScript

I. Introduction

In modern software development, performing code checks has become a vital part of the development process. Static code analysis complements quality assurance techniques, namely manual code review and testing, to provide support for quality assessment of the code [1]–[3]. Tools performing static analysis automatically, most commonly referred to as *automated static analysis tools (ASATs)* [2]–[5], analyze a software without having to run it [6]. Instead, ASATs work directly on a software's code [3], [5]. They take advantage of code representations, e.g., abstract syntax trees, by utilizing them to represent software's code and inspecting them to find patterns that are known to be problematic [5], [7]. The tools define such patterns in the form of rules, which are tool- and language-dependent yet, in general, concerned with detecting coding mistakes, noncompliance with best practices, and violations of coding conventions [3], [7]. If the tool detects that a rule is violated, it generates a warning, alerting developers about potential issues in the code [6], [7].

Developers can benefit significantly from using ASATs as they enable early detection of quality flaws, potential faults, and deviations from coding standards and guidelines. This helps them to reduce defects, prevent inconsistencies, and improve the overall software quality of their projects [1], [2], [4], [8]. Due to these benefits, developers often define ASATs in software repositories and use them daily in several contexts, in particular local programming, code reviews, and continuous integration [2], [3].

With the increasing popularity of JavaScript as the language of choice for software development [9], [10], static analysis for JavaScript-based software has become an active area of research in recent years. Especially considering that its dynamic nature largely contributes to inconsistencies [11] and proneness to coding errors and defects [12]–[14]. In the code of dynamically-typed languages, unexpected behavior can quickly be introduced with simple syntactic or spelling mistakes and can go unnoticed for a long time due to the absence of compilers [11], [15]. Contrarily, in the code of statically-typed languages, such mistakes are addressed at compile time. Hence, software written in dynamically-typed languages can benefit more from ASATs [3]. In the JavaScript community, ASATs performing relatively simple static analysis to flag the aforementioned issues are referred to as *linters* [11], [14], [16]. While prior work has already investigated the adoption of JavaScript linters in practice [11], [14], [17], [18], as well as the usage of ASATs from a long-term perspective in general [2], [3], [5], there is a lack of empirical evidence on how their usage and reports evolve in real-world JavaScript-based software projects throughout their entire development lifespan. To address this, this work conducts an in-depth case study on the repository histories of 15 open-source JavaScript npm packages, focusing on linter usage, changes in configurations, and evolution of generated warnings throughout packages' development.

In summary, the main contributions of this work are:

- We present a case study on linting tools usage and their configuration evolution based on a fine-grained commit-level analysis of 15 JavaScript npm packages.
- We present a case study on linter warning trends characteristics and evolution based on a fine-grained commit-level analysis utilizing warnings generated by the ESLint tool on 15 JavaScript npm packages.
- We build on findings from existing studies by providing a more in-depth, longitudinal perspective on linting tool usage and warnings in the JavaScript ecosystem throughout entire packages' development.

II. Related Work

Existing empirical studies have investigated ASATs for different programming languages from various research perspectives. Researchers have addressed their perception and usage by practitioners [2]–[4], [6], [11], [14], [18], [19], their impact on software quality [5], [7], [20], automatic project-specific customization of configurations [1], [19], capturing of warning resolution patterns [4], [21]–[23], and their integration into development pipelines [8].

In recent years, several researchers have addressed ASATs specifically for JavaScript code. Tómasdóttir et al. [11] interviewed 15 developers responsible for linter configurations in JavaScript projects, focusing on why they use linters, how they configure them, and what challenges they face. The study was later extended [14] with a survey of 337 developers from the JavaScript community, further discussing the matter, and an empirical analysis of 9,500 ESLint configuration files, exploiting the most common linter configuration patterns. Santos et al. [17] investigated bad coding practices in 31 JavaScript systems, while Campos et al. [16] analyzed linter rule violations in 336,000 JavaScript code snippets published on Stack Overflow. Rafnsson et al. [15] investigated the adoption of linters for finding and preventing security vulnerabilities. Ueda et al. [19] proposed a method to automatically customize linter rules to adjust to software projects' code. While these studies have researched various aspects of the usage of JavaScript linters in practice, to the best of our knowledge, no study has investigated how their usage, configurations, and violations evolve throughout a project's development lifetime; the gap we aim to address with this work. Although others have investigated evolutionary aspects of code quality assurance in the context of JavaScript projects [13], [18], [24], [25], this has not been yet addressed from the point of view of linter configurations and reports.

There have been some similar retrospective studies conducted to ours. Beller et al. [3] investigated the use of nine ASATs and their configurations in 168,214 projects written in Java, JavaScript, Ruby, and Python. The results showed that 59% of the projects relied on at least one ASAT and that 80% of configuration files never changed. For the ones that did, the changes were small and occurred in close time intervals. These findings were confirmed in a later study by Vassallo et al. [2], where 56 developers were surveyed. Half of the participants reported that they configure ASATs at the project's kick-off only, yet, 20% declared they modify configurations monthly. An additional inspection of 176 open-source projects written in Java, JavaScript, Ruby, and Python revealed that, in general, 66% of the analyzed projects defined at least one ASAT in their repository, while the percentage is even higher in the case of JavaScript (81%). Another similar study was conducted by Trautsch et al. [5]. The authors investigated the evolution of warnings generated by the PMD tool on 54 Java projects throughout their development. The results showed that the warnings' sum increased over the years in most projects. However, as the number of warnings was found to be correlated with the size of the projects, the density of warnings calculated per logical lines of code in a project was found to be decreasing. Thus, they conclude that, on average, the overall general quality of code is improving over time if measured by warning density. Our study differs from these longitudinal studies in three key aspects: (i) We focus on JavaScript software projects only, which may yield different results because of the differences in ASATs for statically-typed and dynamically-typed languages. (ii) We focus on a smaller set of projects, which allows us to provide a more in-depth analysis. (iii) As projects are developed iteratively, we take a closer look at the evolution of linter usage and warnings with regard to commits, release versions, and calendar years.

III. Study Design

A. Objectives and Research Questions

This work sets out to address the following research questions (RQs):

- *RQ1* How does linter usage evolve throughout development in JavaScript packages?
  - *RQ1.1* How often and when does a change in a linter tool configured for a package occurs?
  - *RQ1.2* How often and why do linter configurations for a package change?
- *RQ2* How do linter warnings evolve throughout development in JavaScript packages?
  - *RQ2.1* Are linter warnings increasing throughout development of a package?
  - *RQ2.2* Are there differences in warning trends between distinct groups of warnings (i.e., possible logic errors, best practices, and style-related warnings)?

B. Case Selection

To address the research questions, 15 packages were selected randomly from the npm registry [26] to be included in the case study. To ensure inclusion of the relevant case subjects, the selection process was guided by the following criteria: programming language (package is JavaScript-based), code availability (code of the package is publicly available on GitHub), repository activity (package's repository contains at least 250 commits), and maturity (package has released at least three major versions).

C. Data Collection

For every case subject, the source code and the list of commit history along with their metadata (contributor, date, commit message, etc.) were collected from a GitHub repository. Commits in the repositories are commonly made simultaneously in the main branch and several sub-branches, which eventually merge with the master branch. Thus, considering all commits made while ordering them by date could result in unwanted jumps in data as several

TABLE I: JavaScript npm packages selected in the case study with the characteristics of included commits.

PACKAGE	NPM WEEKLY DOWNLOADS*	NUMBER OF COMMITS	NUMBER OF RELEASES	RANGE OF RELEASES	TIME INTERVAL OF RELEASES	GITHUB REPOSITORY**
ACORN	70,168,567	966	106	0.0.1–5.7.4	Sep., 2012–Mar., 2020	*acornjs/acorn*
ASYNC	54,083,018	1,012	85	0.1.0–3.2.2	Jun., 2010–Jan., 2022	*caolan/async*
BABEL LOADER	14,647,066	342	63	0.1.0–8.2.2	Oct., 2014–Oct., 2021	*babel/babel-loader*
CHAI	5,047,917	768	79	0.0.1–4.3.5	Dec., 2011–Jan., 2022	*chaijs/chai*
CSS LOADER	17,848,320	663	138	0.1.0–6.7.0	Apr., 2012–Mar., 2022	*webpack-contrib/css-loader*
GULP	1,448,092	755	74	0.0.1–4.0.1	Jul., 2013–May, 2019	*gulpjs/gulp*
HEXO	18,546	1687	136	0.0.1–6.0.0	Sep., 2012–Jan., 2022	*hexojs/hexo*
INQUIRER	25,326,691	499	87	0.0.0–6.0.0	May, 2013–Jun., 2018	*SBoudrias/Inquirer.js*
JS YAML	47,644,947	1,073	75	0.0.1–4.0.0	Oct., 2011–Apr., 2021	*nodeca/js-yaml*
MOCHA	6,408,988	2,574	184	0.0.0–9.2.0	Aug., 2011–Feb., 2022	*mochajs/mocha*
NODEMAILER	1,918,883	727	223	0.0.0–6.7.1	Jan., 2011–Nov., 2021	*nodemailer/nodemailer*
QS	65,720,585	622	84	0.0.0–6.9.7	Jul., 2014–Jan., 2022	*ljharb/qs*
RIOT	5,582	2,697	202	0.9.10–6.1.1	Feb., 2014–Jan., 2022	*riot/riot*
TAPE	663,533	696	133	0.0.0–5.5.1	Nov., 2012–Feb., 2022	*substack/tape*
WS	56,806,584	1,481	155	0.0.1–8.5.0	Nov., 2011–Feb., 2022	*websockets/ws*

* Data was collected from the npm package registry on Feb. 27, 2022. ** Packages' repositories are available at *https://github.com/*.

different variations of the source code exist at the same time. As this work aims to analyze the evolutionary trends, a single sequence of commits with consecutively made commits over time was selected instead, following the procedure conducted by Trautsch [5]. We started from the oldest commit available on the master branch, moving up in the parent-child-structured commit history graph by selecting the first parent of each commit, finishing at the newest available commit on the master branch. For each commit, package version specified in the `package.json` file was extracted. Versions were represented using semantic versioning in the form of *Major.Minor.Patch*, e.g., 2.0.1 (note that pre-release identifiers were discarded). Commits that did not include the package version were omitted. A more detailed overview of packages and their development history included in the case study is presented in Table I.

1) RQ1: To capture the usage and configurations of linters in each package, the commits were mined. We limited our analysis to four linting tools, i.e., ESLint, JSLint, JSHint, and JSCS, the most commonly used linters in the JavaScript community based on prior conducted study [3]. To answer *RQ1.1*, the introduction or removal of any of the four analyzed linters was extracted with the help of the `git log` command and captured changes to specified *"dependencies"* or *"devDependencies"* fields in the package's `package.json` file. To answer *RQ1.2*, the introduction, change, or removal of configurations files of any of the four analyzed linters were captured using the `git log` command and detected changes to configurations files. For each linter, relevant files where their configurations can be specified were considered – for ESLint configurations are placed inside *"eslintConfig"* field in the `package.json` file or in the `.eslintrc.{js, cjs, json, yaml, yml}` file, for JSHint configurations are placed inside *"jshintConfig"* field in the `package.json` file or in the `.jshintrc` file, and for JSCS configurations are placed in the `.jscsrc.{js, json, yaml}` file. The two processes were fully automated using Bash and Python3 scripts. To ensure the correctness of the data collection processes, manual validation was conducted.

2) RQ2: To answer *RQ2*, each commit was traversed and analyzed using ESLint v8.10.0. We chose the ESLint tool as it is the most commonly used linter in the JavaScript ecosystem [2]. In the static analysis, files with non-production code were ignored – directories with files containing test code (`test`, `tests`, `coverage`), documentation files (`doc`, `docs`, `examples`), distribution files (`dist`, `releases`, `min`), etc. ESLint was run with a preset of rules *eslint:all*, which includes 262 rules regarding possible problems (56 rules), best practice suggestions (144 rules), and rules related to layout and formatting of the code (62 rules). With this data collection process, we extracted warning count per commit on two granularity levels: warning count of each of the three groups of distinct types of warnings and total warning count. In addition, for counting lines of code, CLOC v1.92 was used. Again, scripts were used to automate the process and manual inspection was done to check for possible anomalies.

D. Data Analysis

1) RQ1: Qualitative data analysis and simple descriptive statistics were performed on the collected data.

2) RQ2: Additional metric – warning density – representing a ratio between warning count and software size was calculated for each commit as the count of warnings per 1,000 lines of code. To assess the normality of the data distribution, the Kolmogorov-Smirnov test was used. The non-parametric Spearman's correlation coefficient (ρ) was then utilized to determine the correlation between software size and warning count. Trend analysis was done using the Mann-Kendall test, a non-parametric test for monotonic trends, to analyze how the warnings evolve. Sen's slope estimator (β) was used to estimate the magnitude of the trends. For assessing the differences in trends between groups of warnings, the Marginal homogeneity test and the Wilcoxon signed-rank test were used.

IV. Results and Discussion

A. RQ1: Linter usage evolution

Out of 15 case subjects, only 1 did not include any linter throughout its development. The remaining 14 defined at least one linter early in development (6 packages defined (first) linter within a year from kick-off) or over the course of development. Only one linter was configured throughout development in 5 packages, two different linters were configured in 3 packages, three linters in 5 packages, and four linters in 1 package. The configuration of multiple linters in a package repository was done either as separately configured linters in mutually exclusive time periods or a combination of linters in the same period. ESLint and JSLint were mainly used alone; any overlap with other linters was short and a part of a transitional period. In contrast, JSCS was often combined with JSHint. The inclusion of JSLint, JSHint, or JSCS in packages' repositories was relatively short, on average 2, 1.3, and 1.6 years, respectively. ESLint is the current linter specified in the latest version of the codebase in 14 packages and has been up to the latest version, on average, included for 5.1 years. Among all packages that replaced a linter, we observe two general movements: from JSLint to JSHint, and from JSHint to ESLint. In addition, all migrations to ESLint happened at a similar time, between April 2015 and October 2016. Moreover, among packages that configured only one linter, ESLint was included in repositories between June 2015 and May 2018. This shows that packages generally move to linters which provide more rules and configuration options, confirming the assumption already made by Kavaler et al. [18]. More detailed, package-specific results are presented in Appendix A.

Out of 28 configuration files related to linters specified by our case subjects, 25 were modified after their introduction. On median, 15 changes overall, i.e., 1.9 changes per year of linter usage, were made to configurations throughout the development of a package. Although previous studies have shown that ASATs are commonly adopted without modifications to default configurations [3], a higher rate of configuration changes was found for JavaScript linters. However, most changes were minor, as 91% of these changes were smaller changes to global options or changes to a few rules. Only 9% could be considered major, where a complete set of rules was changed, which could greatly affect developers' tool usage. In addition, changes related to modifying linter configuration files represent, on median, only 1.4% of all commits made to our case subjects (without considering commits related to the introduction and removal of these files). It is interesting to note that changes to configurations were often made by the same contributors – on median, by only 5.5% of all contributors. Most changes (57.1%) happened because of addition, deletion, or adjustment to rules. Some changes (33.7%) were made to modify global configuration options, including global variables, environments, and ignore patterns. A few changes (9.2%) occurred due to organizational or stylistic changes to configuration files.

> *RQ1.1* Out of 15 packages, 9 configured more than one linter throughout their development. The changes in linters are scarce and predominantly due to general movements to more advance linters, e.g., from JSHint to ESLint. *RQ1.2* Most linter configurations (89.3%) were modified at least once after their introduction to a package repository. Yet, these changes did not happen often. When they did, 91% of them were minor and mainly aimed to add, delete, or adjust rules.

B. RQ2: Linter warnings evolution

In Table II, the results of the trend analysis are presented. In 13 out of 15 case subjects, the number of warnings has an overall monotonic upward trend throughout development. From a more short-term perspective, where commits of each case subject are grouped by major release version, an increasing trend was found in 74.3% of release versions. In a study by Trautsch et al. [5], the absolute number of warnings per commit was found to be positively correlated with lines of code in Java projects (ρ=0.72, p=0.00). Our study on JavaScript packages showed an even stronger positive correlation (ρ=0.95, p=0.00). Thus, as packages grow in terms of code size, the number of generated linter warnings generally increases. Investigating warning evolution through warning density revealed that 13 subjects out of 15 were found to have a consistent downward warning density trend throughout development. This shows that, in general, packages are improving in terms of generated warnings per line of code. By grouping commits of each case subject by major release version, a decreasing trend was found in 44.6% of release versions. There is no consensus across all case subjects that would indicate similar trends in release versions.

For most subjects, the three groups of warnings had the same warning trends, i.e., if the warning count trend was upward for warnings related to best practice suggestions, the same was true for styling-related warnings. The group of warning with possible errors was found to be the most different among the three as differences to the other two groups were found in 6 and 3 subjects for warning count and density trend, respectively. The only statistically significant difference in overall trends was found in terms of warning count between warnings regarding possible errors and best practice suggestions (p=0.02), where more warnings related to possible errors decreased rather than increased compared to the other group. In Table III, the pairwise analysis of the differences in trends' slopes between groups of warnings is presented. The magnitude of the trend is significantly different between most pairs. The highest positive slope of the warning count trend was found for warnings related to stylistic issues (M=8.23), followed by best practices (M=1.2) and possible logic errors (M=0.00). Similarly, the highest negative slope of the warning density trend was found for stylistic issues (M=-0.4), followed by best practices (M=-0.25) and possible errors (M=-0.02). This shows that, in general,

TABLE II: Monotonic warning trends overall and per major releases.

Package	Warning Count Trend	Warning Density Trend	Warning Density Trend Per Major Releases									
			R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
ACORN	pos. (Z=41.83, β=8.76)	neg. (Z=-32.66, β=-2.83)	neg.	pos.	pos.	pos.	neg.	neg.				
ASYNC	pos. (Z=27.09, β=14.59)	neg. (Z=-16.43, β=-0.20)	neg.	neg.	pos.	n.s.						
BABEL-LOADER	pos. (Z=20.98, β=2.07)	neg. (Z=-8.86, β=-2.63)	neg.	/	/	/	n.s.	n.s.	neg.	neg.	pos.	
CHAI	pos. (Z=38.08, β=41.12)	pos. (Z=26.96, β=2.22)	pos.	neg.	n.s.	pos.	neg.					
CSSLOADER	pos. (Z=32.69, β=6.95)	neg. (Z=-26.21, β=-3.24)	neg.	pos.	neg.	pos.	neg.	n.s.	neg.			
GULP	neg. (Z=-18.75, β=-0.42)	neg. (Z=-19.17, β=-1.42)	pos.	/	pos.	neg.	neg.					
HEXO	pos. (Z=17.13, β=9.80)	neg. (Z=-15.91, β=-0.45)	pos.	neg.	pos.	neg.	pos.	pos.	/			
INQUIRER	pos. (Z=18.18, β=7.48)	neg. (Z=-12.63, β=-0.60)	n.s.	neg.	n.s.	pos.	pos.	n.s.	/			
JSYAML	n.s. (Z=1.33, β=1.58)	neg. (Z=-8.10, β=-0.42)	pos.	pos.	pos.	neg.	/					
MOCHA	pos. (Z=26.51, β=8.09)	neg. (Z=-13.35, β=-0.47)	neg.	n.s.	neg.	n.s.	neg.	neg.	neg.	neg.	neg.	neg.
NODEMAILER	pos. (Z=11.62, β=8.31)	neg. (Z=-23.51, β=-2.96)	neg.	pos.	neg.	neg.	neg.	/	pos.			
QS	pos. (Z=30.50, β=3.05)	pos. (Z=6.87, β=0.23)	neg.	pos.	pos.	n.s.	n.s.	pos.	n.s.			
RIOT	pos. (Z=52.32, β=32.35)	neg. (Z=-27.89, β=-0.44)	n.s.	n.s.	neg.	pos.	pos.	pos.	pos.			
TAPE	pos. (Z=34.57, β=2.15)	neg. (Z=-30.23, β=-0.65)	neg.	neg.	neg.	neg.	neg.	neg.				
WS	pos. (Z=3.11, β=0.67)	neg. (Z=-33.86, β=-1.04)	pos.	neg.	neg.	pos.	n.s.	pos.	neg.	neg.	neg.	

pos.=positive ($p\leq$0.05) neg.=negative ($p\leq$0.05) n.s.=not significant ($p>$0.05) /=cannot be determined ($n<$8)

warnings related to code style increased the most in terms of absolute count throughout development and decreased the most in terms of warning count per line of code.

TABLE III: Pairwise differences in overall warning trends' slopes between the three groups of warnings.

	Warning Count Slope Difference	Warning Density Slope Difference
Errors–Best practices	Z=-3.29, p=0.00*	Z=-2.22, p=0.03*
Best practices–Styling	Z=-3.01, p=0.00*	Z=-1.02, p=0.31
Styling–Errors	Z=-3.24, p=0.00*	Z=-2.50, p=0.01*

* Statistically significant results ($p\leq$0.05)

RQ2.1 Warnings are generally increasing throughout development with package size. However, warnings per line of code are decreasing in most cases. *RQ2.2* There are rarely any differences in trends among distinct groups of warnings, though the magnitude of their trends significantly differs.

V. Conclusion

In this work, we investigated software repositories of 15 JavaScript-based npm packages to improve our understanding of how linter usage and warnings evolve in the real world. There are several additional aspects studies could address in the future. Firstly, a similar study on linter warnings could be conducted, but rather than using the same preset rules for all projects, project-defined rules could be used to investigate if this study setting would yield different conclusions. Secondly, a survival analysis of linter warnings would provide some additional understanding of when a particular rule violation is introduced, how long it persists in the code, and when it is resolved. Furthermore, future research could investigate the relationship between linter warnings and external quality factors.

A. Threats to Validity

Findings are based on a limited set of JavaScript packages, and further studies are needed to verify the generalizability of these findings. To reduce the threat related to selection bias, our case subjects consist of a diverse set of projects in terms of size and domains used. All subjects are open-source. As such, its generalizability to closed-source projects might be limited. Though, the open-source nature of the included subjects facilitates the study's replicability. In the context of *RQ1*, the scope of the targeted linters is limited to the four most popular linters. We expect similar results to be observed even if more linters were included. However, additional research is required to confirm this. We noticed that several packages rely on external presets of rules; thus, all changes in rules followed by a package might not be reflected by the package's repository itself. As our study does not consider changes not included in a package's repository, this is a limitation of our study. In the context of *RQ2*, we observe linter warnings reported by one linter only – ESLint. Replication studies are needed to confirm and generalize the findings with other linters.

Acknowledgment

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0057).

References

[1] F. Zampetti, S. Mudbhari, V. Arnaoudova, M. Di Penta, S. Panichella, and G. Antoniol, "Using code reviews to automatically configure static analysis tools," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–30, 2022.

[2] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, 2020.

[3] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, 2016, pp. 470–481.

[4] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed? a closer look at realistic usage of sonarqube," in *IEEE/ACM 27th International Conference on Program Comprehension*, 2019, pp. 209–219.
[5] A. Trautsch, S. Herbold, and J. Grabowski, "A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects," *Empirical Software Engineering*, vol. 25, no. 6, pp. 5137–5192, 2020.
[6] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 332–343.
[7] A. Trautsch, S. Herbold, and J. Grabowski, "Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction," in *IEEE International Conference on Software Maintenance and Evolution*, 2020, pp. 127–138.
[8] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *IEEE/ACM 14th International Conference on Mining Software Repositories*, 2017, pp. 334–344.
[9] GitHub, "The 2021 state of the octoverse," 2022, [Online]. Available: https://octoverse.github.com/.
[10] StackOverflow, "2021 developer survey," 2022, [Online]. Available: https://insights.stackoverflow.com/survey/2021.
[11] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, "Why and how javascript developers use linters," in *32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 578–589.
[12] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn, "Javascript errors in the wild: An empirical study," in *IEEE International Symposium on Software Reliability Engineering*, 2011, pp. 100–109.
[13] D. Johannes, F. Khomh, and G. Antoniol, "A large-scale empirical study of code smells in javascript projects," *Software Quality Journal*, vol. 27, no. 3, pp. 1271–1314, 2019.
[14] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, "The adoption of javascript linters in practice: A case study on eslint," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 863–891, 2020.
[15] W. Rafnsson, R. Giustolisi, M. Kragerup, and M. Høyrup, "Fixing vulnerabilities automatically with linters," in *Network and System Security: 14th International Conference*. Berlin, Heidelberg: Springer-Verlag, 2020, pp. 224–244.
[16] U. Ferreira Campos, G. Smethurst, J. P. Moraes, R. Bonifácio, and G. Pinto, "Mining rule violations in javascript code snippets," in *IEEE/ACM 16th International Conference on Mining Software Repositories*, 2019, pp. 195–199.
[17] A. Santos, M. Valente, and E. Figueiredo, "Using javascript static checkers on github systems: A first evaluation," in *Proccedings of the 3rd workshop on software visualization, evolution and maintenance*, 09 2015, pp. 33–40.
[18] D. Kavaler, A. Trockman, B. Vasilescu, and V. Filkov, "Tool choice matters: Javascript quality assurance tools and usage outcomes in github projects," in *IEEE/ACM 41st International Conference on Software Engineering*, 2019, pp. 476–487.
[19] Y. Ueda, T. Ishio, and K. Matsumoto, "Automatically customizing static analysis tools to coding rules really followed by developers," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2021, pp. 541–545.
[20] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study," *Journal of Systems and Software*, vol. 170, p. 110750, 2020.
[21] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the apache ecosystem?" in *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 153–163.
[22] N. Imtiaz, B. Murphy, and L. Williams, "How do developers act on static analysis alerts? an empirical study of coverity usage," in *IEEE 30th International Symposium on Software Reliability Engineering*, 2019, pp. 323–333.
[23] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2021.
[24] D. Mitropoulos, P. Louridas, V. Salis, and D. Spinellis, "Time present and time past: Analyzing the evolution of javascript code in the wild," in *IEEE/ACM 16th International Conference on Mining Software Repositories*, 2019, pp. 126–137.
[25] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in javascript projects," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 294–305.
[26] "npm registry," 2022, [Online]. Available: https://www.npmjs.com/.

APPENDIX

A. Linter tool usage evolution of each case subject

The evolution of linter tool usage in each package included in the case study was found to be as follows:

- ACORN: ESLint was added in v4.0.11 (Mar., 2017).
- ASYNC: JSLint (using *nodelint* package) was added in v0.1.16 (Jan., 2012). The linter was replaced by JSHint and JSCS in v0.9.2 (May, 2015). In v2.0.0 (Jun., 2016), the two linters were replaced by ESlint.
- BABEL LOADER: JSHint and JSCS were configured in v5.1.3 (Jun., 2015). Migration to ESLint (using *babel-eslint* package), was done in v6.2.7 (Nov., 2016).
- CHAI: No linter has been configured in the package's repository throughout the entire development history.
- CSS LOADER: ESLint was added in v0.26.1 (Jan., 2017).
- GULP: In v2.1.0 (Nov., 2013), JSHint was configured. In v3.8.11 (Feb., 2015), JSCS was added to the package. Shortly after that, in v3.9.0 (Sep., 2015), ESLint replaced JSHint. Yet, JSCS was not removed until v4.0.0 (Avg., 2018).
- HEXO: JSHint (using *grunt-contrib-jshint* package) was added in v2.4.5 (Feb., 2014). In v3.1.1 (Sep., 2015), ESLint and JSCS were included instead of JSHint. JSCS was later removed in v3.4.4 (Jan., 2018), whereas ESLint is used up to the latest version.
- INQUIRER: At development start, in v0.0.0 (May, 2013), JSHint (using *grunt-contrib-jshint* package) was configured. In v0.12.0 (Mar., 2016), it was replaced by ESLint.
- JS YAML: JSLint was specified early in development, in v0.3.1 (Dec., 2011). However, it was removed in v0.3.7 (Jul., 2012) and replaced by JSHint. Note that JSHint was not defined in the package's `package.json` file. However, the migration to JSHint was reflected by the commit message (*"Replace jslint with jshint"*) and the establishment of JSHint configuration files. In v3.2.7 (Apr., 2015), JSHint was replaced by ESLint.
- MOCHA: In v2.2.5 (Jun., 2015), ESLint was added.
- NODEMAILER: JSHint (using *grunt-contrib-jshint* package) was added in v1.0.0 (Jun., 2014) and removed in v2.0.0 (Dec., 2015) when ESLint (using *grunt-eslint* package) was configured.
- QS: At project kick-off, in v0.0.0 (Jul., 2014), JSHint was configured. However, it was removed a day later in v1.0.4 (Jul., 2014). ESLint was then introduced in v6.0.1 (Dec., 2015).
- RIOT: Shortly after the beginning of development, in v0.9.8 (Feb., 2014), JSHint was specified. In v2.0.9 (Feb., 2015), JSCS was also added. Both linters were replaced by ESLint in v2.0.10 (Feb., 2015).
- TAPE: ESLint was introduced in v4.9.2 (May, 2018).
- WS: In v1.1.0 (Jun., 2016), ESLint was configured.