Rule Violations in JavaScript Open Source Projects on GitHub: An ESLint Perspective

Qinwen Xu
Faculty of Applied Sciences
Simon Fraser University
Burnaby, Canada
qxa13@sfu.ca

Jiawei Zhou
Faculty of Applied Sciences
Simon Fraser University
Burnaby, Canada
jza139@sfu.ca

***Abstract*—What is the quality of JavaScript open-source projects on GitHub? In this paper, we present a quantitative analysis of rule violations in JavaScript projects hosted on GitHub, using ESLint as our primary analytical tool. By leveraging data from real-world projects, this research will provide valuable insights into the quality of GitHub projects, analyzing prevalent code quality issues and adherence to coding standards. We begin by outlining the methodology for selecting 400 JavaScript projects and statically analyzing them with ESLint, a JavaScript linter. We discovered that not a single project in our study was free from rule violations, underscoring the pervasive nature of coding issues in open-source environments. Our analysis highlights that stylistic issues are the most prevalent, accounting for 89.9% of all rule violations, followed by variable-related issues at 5.65%. Interestingly, critical 'Possible Errors' are rare, representing just 0.13% of the total infractions.**

I. Introduction

GitHub has emerged as a central platform for software development, having over 100 million developers and hosting more than 372 million repositories[1]. Its significance lies in providing tools for version control, collaboration, and issue tracking, which are essential in modern software development. GitHub's impact is particularly notable in its ability to bring together developers from around the world, fostering a community where code is not just shared but continuously improved upon. Based on data from GitHut 2.0, JavaScript accounts for 10% of the projects on GitHub, ranking as the fourth most popular programming language on the platform[2]. JavaScript, extensively employed in the development of web and mobile applications, desktop interfaces, and server-side solutions, is a key player in the realm of open-source projects. However, the intrinsic dynamic and flexible attributes have raised concerns about the quality of software within these open-source environments, where adherence to standardized coding practices is often variable. This study aims to deepen the understanding of code quality in JavaScript projects hosted on GitHub, with a particular emphasis on rule violations in open-source projects. Utilizing the GitHub API, we selected a sample of 400 JavaScript projects for our analysis. For the evaluation of these projects, we employed ESLint, a widely recognized open-source linter, to identify possible errors, bugs, and deviations from stylistic coding standards. The primary outcomes of this research are as follows:

- We observed that violations are ubiquitous in our sampled projects. No project is free of rule violations.
- These violations were classified according to five categories defined by ESLint: Best Practices, Variables, Stylistic Issues, Possible Errors, ECMAScript, and Node.js and CommonJs rules. Stylistic Issues emerged as the most common, constituting 89.9% of all violations, followed by issues related to Variables at 5.65%. In contrast, 'Possible Errors', which signify more critical concerns, were found to be remarkably infrequent, accounting for merely 0.13% of the violations. No violations pertaining to Node.js and CommonJs rules were detected in the projects analyzed.
- Our investigation into JavaScript open-source projects on GitHub exposes Stylistic Issues, specifically infractions related to 'indent,' 'semi,' and 'quotes,' as the most pervasive among the error types scrutinized. The prevalence of these violations highlights the difficulties developers face in upholding code uniformity. This insight sets the stage for a deeper exploration into the root causes and potential solutions for enhancing coding standards within open-source projects on GitHub.

II. Background

Linters, as a subset of lightweight static analysis tools, perform straightforward analyses to identify programming errors, promote best practices, and ensure adherence to stylistic coding standards. The foundation of this study is ESLint, which stands as the most widely downloaded and utilized linter for JavaScript projects. ESLint operates based on an Abstract Syntax Tree (AST) search mechanism, scanning for specific code patterns within the nodes of the AST.

ESLint, known for its high degree of customizability[3], does not include a default configuration upon installation. Instead, it offers multiple methods for rule configuration: (1)

specifying individual rules in a configuration file, (2) annotating individual rules with comments within code files, (3) employing a preset configuration, and (4) integrating plugins. In this study, we configured ESLint using its standard rule set, which comprises a comprehensive catalogue of 253 rules. A violation was duly recorded in instances where any of these predefined rules were breached.

III. Methodology

A. Research Questions

In this paper, we answer the following three research questions:

- RQ1: How commonplace are rule violations in JavaScript open-source projects on GitHub?
- RQ2: What is the frequency and distribution of rule violations in these projects?
- RQ3: What are the potential root causes behind these infractions?

To address RQ1, we examine the proportion of projects exhibiting any form of rule violations. Our selection criteria focus on well-known, actively updated, public projects on GitHub to ensure relevance and timeliness. For RQ2, we analyze the occurrences of violations across the six categories, which will be detailed in Section III-C. Lastly, in responding to RQ3, we identify the most frequently violated rules within each category and analyze their root causes, particularly in the context of JavaScript language features.

B. Data Extraction

Sampling JavaScript projects on GitHub Our methodology for sampling JavaScript projects on GitHub involved the use of PyGithub, an interface for the GitHub API, which allowed us to search for open-source JavaScript projects meeting specific criteria. These criteria were established to ensure a comprehensive representation of significant JavaScript projects:

- Public Accessibility: The project must be publicly accessible.
- Originality: Only original projects were considered, excluding forks.
- Primary Language: The project's primary coding language must be JavaScript.
- Popularity: Projects should have accumulated over 100 stars, indicating a level of popularity and community engagement.
- Project Size: The project size should exceed 10MB.
- Maintenance Indicators: Evidence of ongoing maintenance, demonstrated by recent updates.

Based on these criteria, systematic sampling was used to select 400 projects. We then extracted the GitHub URLs of these projects as our raw data source.

Linking the dataset For the analysis of code quality, ESLint was installed with a standard configuration on local machines. A custom script was developed to automate the downloading of projects and the execution of ESLint analysis on each. This script was designed to capture the output data of warnings and errors from ESLint. The standard configuration of ESLint implements a total of 253 rules. Rules are grouped by categories, namely[4]:

- Possible Errors: Rules relate to possible syntax or logic errors in JavaScript code. There are 34 rules.
- Best Practices: Rules relate to better ways of doing things to help developers avoid problems. There are 70 rules.
- Variables: Rules relate to variable declarations. There are 12 rules.
- Node.js and CommonJS: Rules relate to code running in Node.js, or in browsers with CommonJS. There are 11 rules.
- Stylistic Issues: Rules relate to style guidelines, and are therefore quite subjective. There are 87 rules.
- ECMAScript6: Rules relate to ES6, also known as ES2015. There are 31 rules.

ETL Process for ESLint Warnings/Errors In the subsequent stage, we applied Extract, Transform and Load (ETL) processes. This involved categorizing the data according to the specific violation rules and quantifying these violations. The ETL processes were crucial in structuring the data for comprehensive analysis and interpretation in the context of our research objectives. After this procedure, we ended up with 59 million rule violation records.

IV. Results

In the upcoming section, we present our organized findings in line with the earlier research questions. Given the considerable variability in total errors across projects, ranging from around 250 to over 200,000, our method involves calculating the average proportion of a rule violation within each project. This approach aims to reveal the distribution of rule violations in the sample. We investigate what percentage of a project's total errors can be attributed to a specific rule violation. Additional analysis details will be available in our GitHub repository[5], providing insights to address the research question.

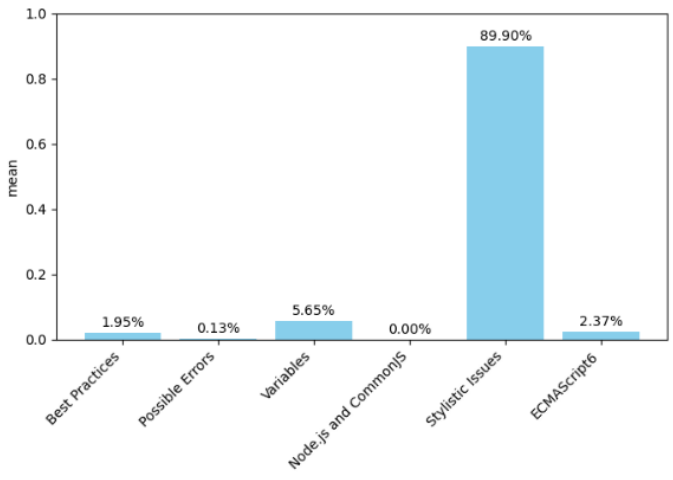

Fig. 1. How frequently an error is Violated across all projects.

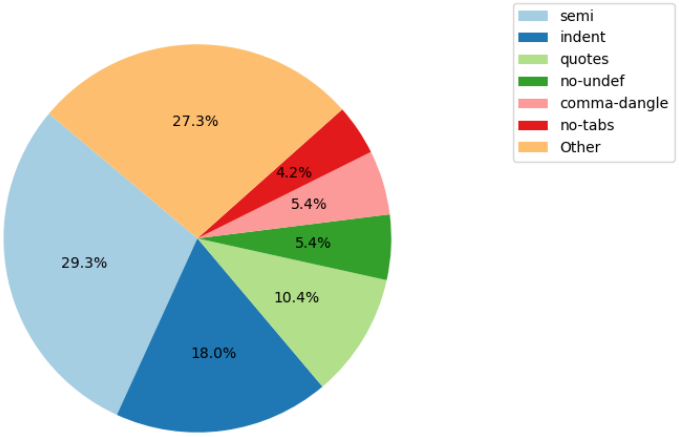

Fig. 2. Rules that contributing more than 70% of violated errors.

RQ1: Stylistic issues stand out as the primary category, comprising 89.9% of errors. The top six rules collectively contribute over 70% across 400 projects, with only 'no-undef' pertaining to variables. Stylistic issues heavily dominate, emphasizing their prevalence in JavaScript projects on GitHub. This analysis illuminates error distribution, highlighting the paramount importance of addressing stylistic issues in the analyzed projects' development practices.

TABLE I
TOP 5 VIOLATED RULES FROM EACH ERROR TYPES

	Stylistic Issues				
Mean(%)	*indent* 21.5	*semi* 15.2	*quotes* 12.3	*comma-spacing* 10.8	*space-infix-ops* 9.6
	Variables				
Mean(%)	*no-undef* 94.8	*no-unused-vars* 4.2	*no-use-before-define* 0.7	*no-shadow-restricted-names* 0.1	*no-undef-init* 0.1
	Possible Errors				
Mean(%)	*no-extra-parens* 55.4	*no-prototype-builtins* 13.0	*no-template-curly-in-string* 7.1	*no-cond-assign* 6.6	*no-func-assign* 3.3
	Best practices				
Mean(%)	*eqeqeq* 15.3	*no-sequences* 13.5	*no-unused-expressions* 13.0	*yoda* 11.6	*no-labels* 10.9
	ECMAScript6				
Mean(%)	*no-var* 77.6	*arrow-spacing* 9.2	*prefer-const* 7.8	*object-shorthand* 4.8	*generator-star-spacing* 0.2

RQ2: We meticulously explore various error types, unveiling commonly violated rules within each domain. Stylistic Issues, notably "indent" and "semi," display distinct patterns, while Variables face breaches in "no-undef," "no-unused-vars," and "no-use-before-define." ECMAScript6 encounters significant infractions in "no-var" and "arrow-spacing." Possible Errors and Best Practices reveal unique, frequently violated rules. This holistic overview illuminates prevalent coding practices and areas where developers commonly deviate from established guidelines in analyzed JavaScript projects. For detailed insights, refer to the accompanying **Table I**.

RQ3: Analyzing the frequent rule violations across different error types provides valuable insights into the potential root causes driving developers to deviate from established coding standards. By examining patterns within Stylistic Issues, Variables, Possible Errors, Best Practices, ECMAScript 6, and Node.js/CommonJS, we gain a comprehensive understanding of the challenges developers face and the factors influencing their coding decisions.

A. Stylistic Issues

In the realm of Stylistic Issues, where consistency is paramount, rules such as 'indent,' 'semi,' 'quotes,' 'comma-spacing,' 'space-infix-ops,' 'comma-dangle,' and 'no-tabs' take center stage. Disparities in indentation, semicolon usage, and quoting styles may arise from mixed development environments, varied tool configurations, and collaborative coding efforts. Challenges with comma spacing, infix operators, trailing commas, and tab usage may stem from manual errors, diverse preferences, and legacy code integration. These findings underscore the complexities of maintaining stylistic uniformity, emphasizing the need for meticulous configurations and standardized practices.

B. Variables

The 'no-undef,' 'no-unused-vars,' and 'no-use-before-define' rules play a pivotal role. 'no-undef' guards against undeclared variables, with violations occurring due to oversight in declarations. 'no-unused-vars' promotes code cleanliness, often violated during refactoring or evolving project requirements. 'no-use-before-define' enforces disciplined variable usage, with infractions stemming from complex code structures or asynchronous operations. Examples highlight the importance of rigorous variable management and nuanced rule enforcement.

C. Possible Errors

Our examination centers on four pivotal rules: 'no-extra-parens,' 'no-prototype-builtins,' 'no-template-curly-in-string,' and 'no-cond-assign.' The 'no-extra-parens' rule aims to eliminate redundant parentheses in code, preventing unnecessary complexity. Violations of this rule may arise from code snippets copied and pasted from other contexts, automated code generation tools, or misunderstandings about operator precedence. The 'no-prototype-builtins' rule focuses on discouraging direct calls to certain Objects. prototype methods, promoting more secure and explicit coding practices. Instances of non-compliance might be observed in legacy codebases or situations where developers are unaware of potential security risks. The 'no-template-curly-in-string' rule guards against using template literals incorrectly by identifying situations where interpolation is unnecessary. Violations can occur due to oversight or the use of template literals without dynamic content. Lastly, the 'no-cond-assign' rule aims to prevent assignment within conditional expressions, avoiding potential logic errors. Violations may stem from typos, mistaken assumptions about JavaScript behaviour, or coding practices from other programming languages. For example, mistakenly using '=' instead of '==' in a conditional statement could trigger a 'no-cond-assign' violation. These examples elucidate the nuanced nature of possible error rule violations and underscore the importance of code review and developer awareness to mitigate such issues.

D. Best Practices

focus on rules like 'eqeqeq,' 'no-sequences,' 'no-unused-expressions,' 'yoda,' and 'no-labels.' The 'eqeqeq' rule advocates for strict equality (=== and !==), enhancing code clarity. Deviations may stem from developers accustomed to loose equality in other languages or legacy codebases. 'no-sequences' discourages the comma operator for clearer code, with non-compliance due to unfamiliarity or programming habits. 'no-unused-expressions' detects and warns against unused expressions, ensuring cleaner code. Violations may arise from copied code, incomplete refactorings, or syntax misunderstandings. The 'yoda' rule advises against Yoda conditions (e.g., if (5 == x)), with deviations from conventional syntax due to different coding conventions or legacy codebases. Lastly, 'no-labels' discourages label use for improved readability. Violations may occur when developers are unfamiliar with this best practice or when adapting code from environments permitting label usage. These examples highlight the rationale behind best practice rules, emphasizing the need for awareness and adherence to coding conventions.

E. ECMAScript6

ECMAScript 6 rules, including 'no-var,' 'arrow-spacing,' 'prefer-const,' and 'object-shorthand,' advocate modern JavaScript features. Developers may resist moving away from 'var' due to habit or transitioning challenges. Consistent spacing in arrow functions, preferring 'const,' and using object shorthand face deviations from inconsistent practices or unfamiliarity. Embracing ES6 features improves code quality and readability.

F. Node.js and CommonJS

The lack of errors in Node.js and CommonJS categories across 400 projects, covering rules like 'callback-return' and 'global-require,' results from key factors. Node.js projects adhere closely to best practices due to the asynchronous nature, minimizing violations in callback-related rules ('callback-return,' 'handle-callback-err'). The absence of 'no-buffer-constructor' errors indicates alternative buffer handling methods, reflecting heightened security awareness. Errors in 'no-process-exit' and 'no-sync' are absent, suggesting a conscientious avoidance of synchronous operations, aligning with Node.js best practices. No violations in 'no-restricted-modules' underscore a commitment to contemporary and secure development. In summary, the absence of errors demonstrates developers' strong adherence to Node.js best practices and awareness of challenges in synchronous operations, callback handling, and module usage.

V. Limitations

Several limitations of this study warrant mention. First, our analysis exclusively utilized ESLint, one of the most widely recognized open-source linters. However, due to the inherent nature of static analysis, ESLint is not capable of detecting every type of code violation. Furthermore, our reliance on ESLint's standard configuration, which encompasses 253 rules, might not encompass the complete spectrum of potential code issues. Second, while GitHub hosts millions of JavaScript projects, our study was confined to a sample of 400, selected based on specific filtering criteria. This sampling approach, although systematic, may not fully represent the vast diversity of JavaScript projects on the platform.

VI. Related work

JavaScript, a crucial programming language, is central to project success, with ESLint serving as a prominent static code analysis tool for ensuring code consistency. Tómasdóttir et al.'s ESLint case study, combining interviews, configuration file analysis, and a developer survey, reveals developers prioritize linters for error prevention over stylistic nuances[6]. Campos et al.'s insights into JavaScript snippet rule[7] violations align with our findings, emphasizing an average of 11.9 violations per snippet, predominantly stylistic. Despite widespread violations, only 0.1% pertain to potential errors, with minimal instances found in GitHub projects. While our project differs by directly assessing rule violations within GitHub projects, cross-referencing with Campos et al.'s approach may enhance our analysis, providing a more comprehensive understanding of JavaScript code rule violations.

VII. Conclusion

In conclusion, our examination of JavaScript open-source projects on GitHub reveals that Stylistic Issues, notably violations related to 'indent,' 'semi,' and 'quotes,' are the most prevalent among the error types analyzed. The widespread occurrence of these violations underscores the challenges developers encounter in maintaining code uniformity. To address this, we recommend fostering awareness and adherence to coding conventions, emphasizing regular code reviews, integrated linters, and automated formatting tools. These practices can significantly mitigate violations in Stylistic Issues, promoting a consistent coding style across projects.

In addressing these issues during development or debugging, developers encountering Stylistic Issues should prioritize reviewing and aligning their code with established coding conventions. Automated tools and linters can be employed to catch and rectify these violations early in the development process. For Variable-related issues, developers are encouraged to focus on comprehensive variable management, ensuring proper declaration and usage throughout the codebase. When dealing with ECMAScript 6-related challenges, developers should prioritize understanding and embracing modern JavaScript features like 'let' and 'const' over 'var,' consistent spacing in arrow functions, and adopting object shorthand syntax. This proactive approach to coding practices and rule adherence can significantly contribute to improved code quality and readability, minimizing errors and enhancing overall project maintainability.

References

[1] GitHub. https://en.wikipedia.org/wiki/GitHub, visited 2023-12-03.

[2] GitHut2.0. https://madnight.github.io/githut/#/pull$_r equests/2023/3, visited 2023-12-03.

[3] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, “Why and how JavaScript developers use linters,” 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 2017, pp. 578-589, doi: 10.1109/ASE.2017.8115668.
[4] Rules. https://denar90.github.io/eslint.github.io/docs/rules/, visited 2023-12-03.
[5] GitHub repository. https://github.com/KevinXu17/982.git visited 2023-12-12.
[6] 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, Aug. 2020, doi: 10.1109/TSE.2018.2871058.
[7] U. Ferreira Campos, G. Smethurst, J. P. Moraes, R. Bonifácio, and G. Pinto, “Mining Rule Violations in JavaScript Code Snippets,” 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 2019, pp. 195-199, doi: 10.1109/MSR.2019.00039.