Decoding Code Quality: A Software Metric Analysis of Open-Source JavaScript Projects

Suzad Mohammad[1,*,a], Abdullah Al Jobair[2,*,b] and Iftekharul Abedeen[2,*,c]

[1] *International University of Business Agriculture and Technology, Dhaka, Bangladesh*

[2] *United International University, Dhaka, Bangladesh*

Keywords: Code Quality, Open-Source Software, JavaScript, Software Metric, Cyclomatic Complexity, Cognitive Complexity, Maintainability, Code Smell, Code Duplication, Project Size, Developer Experience, GitHub.

Abstract: The popularity of web-based solutions has seen rapid growth in the last decade, which has raised the demand for JavaScript (JS) usage in personal projects and enterprise solutions. While the extensive demand for JS has elevated, studies have yet to be done on how JS development follows the rules and guides for writing code to meet quality standards. Consequently, we choose to investigate the practice of JS on different project sizes, the developers' experience, and their impact on code quality and development. To achieve this goal, we perform the code quality analysis of 200 open-source JS projects from GitHub on 10 code quality metrics. We design our research study to examine the influence of project size on issue density, find relationships among 10 code metrics, how code quality changes with developer experience, and determine the capabilities of existing source code evaluation tools. Our findings reveal that issue density decreases with increasing developer experience and project size. In addition, our quantitative study suggests that with the increase in project size and line of code (LOC), project maintainability decreases, leading to more issues such as errors, complexity, code smell, and duplication. However, as developers become more experienced, they face fewer coding challenges, enhance code quality, and reduce code smell per line of code (LOC). Our study also offers valuable insights into the capabilities of the 6 tools mentioned above to advance code evaluation practices.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

In recent years, the popularity of web-based solutions and web apps has grown immensely, and JavaScript (JS) has also seen enormous growth and popularity. As the popularity of JS is gaining more and more foothold, it is also transcending its use within the development of only web-based solutions (Jasim, 2017) (Brito et al., 2018). We can find many solutions from desktop IDE to client applications and mobile apps in significant platforms like Android and IOS that are developed with JS.

As per the official GitHub survey of 2023[1], JS continues to hold its position as the most widely used programming language, retaining its position as the top-ranked language since 2014. Not only GitHub but also StackOverflow, the most prominent Question-Answer (Q/A) website (Al Jobair. et al., 2022), claims JS as the top programming language for eleven consecutive years in their developer survey 2023[2]. All of this gave us the zeal to investigate JavaScript-oriented works as it is currently the most widely used programming language. It allows us to take a look at different development paradigms through the observation of one language. As JS is being brought into more places, many more people are picking up this language to develop their solutions. However, limited studies have been conducted on how JS development follows the principles and code guidelines to achieve quality standards.

Therefore, our goal is to investigate how JavaScript is practiced with projects of different sizes, the developers' experience, identify the relationships among different code metrics, and how each metric affects the development and code quality. To achieve our goal, we structure our research study into the following three research questions (RQ):

[a] https://orcid.org/0009-0004-0161-7296
[b] https://orcid.org/0000-0002-1530-1733
[c] https://orcid.org/0009-0003-7986-5954

[*] These authors contributed equally to this work.

[1] https://tinyurl.com/mvttxecw

[2] https://survey.stackoverflow.co/2023/#technology-most-popular-technologies

Mohammad, S., Al Jobair, A. and Abedeen, I.
Decoding Code Quality: A Software Metric Analysis of Open-Source JavaScript Projects.
DOI: 10.5220/0012618800003687
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, pages 63-74
ISBN: 978-989-758-696-5; ISSN: 2184-4895
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

RQ1: How does project size influence issue density?
RQ2: What are the relations among different code quality metrics, and how do they influence each other?
RQ3: How does code quality change with developer experience?

To answer the research questions, we collect and investigate 200 open-source JS projects from GitHub repositories based on project size. To ensure the diversity of projects, we define the term *"Project Size"* based on three factors - Lines of Code (LOC), number of contributors, and number of commits. These factors collectively contribute to a comprehensive understanding of the development landscape. Moreover, a total of 380 developers' profiles obtained from selected 200 projects are classified into 3 groups based on developer experience. The *Developer Experience* is formulated by incorporating the number of forks, number of stars, number of followers, and activity graph of a GitHub profile. Each factor plays a crucial role in delineating the characteristics and dynamics of open-source JS projects. Furthermore, a thorough quantitative analysis is conducted to assess project metrics.

Our analysis has studied the relationship among 10 code quality metrics: LOC, cyclomatic complexity, cognitive complexity, code smell, code duplication, issue density, maintainability, lint error, code quality, and average estimated error. Our observations reveal that with increased experience, developers encounter fewer code-related challenges and improve code quality and maintenance, resulting in decreased code smell per LOC. As project size and LOC increase, the maintainability of projects decreases while problems like errors, complexity, code smell, and code duplication increase.

Our findings will allow developers to make informed decisions concerning the interplay between experience levels and encountered challenges, ultimately fostering improved code practices. Moreover, by providing insights about the usage of existing analyzer tools, our study equips researchers with valuable information for the advancement of code evaluation practices. Through these contributions, our findings aim to foster continuous improvement in both practical development approaches and scholarly endeavors within the software engineering landscape.

The rest of the paper is structured as follows. We introduce state of the art in Section 2 and the methodology in Section 3. Section 4 presents the results and analysis of our study, answering RQ1, RQ2, and RQ3. Section 5 discusses the threats to validity, followed by the conclusion in Section 6.

2 STATE OF THE ART

Scholars and professionals have engaged in conversations about the impact of code quality analysis in open source software projects (Molnar and Motogna, 2022) (Hussain et al., 2021) (Chren et al., 2022) (Borg et al., 2023), particularly regarding JavaScript projects code quality (Zozas et al., 2022) (Abdurakhimovich, 2023). To establish the foundation for our methodology and structure the research questions, we begin by presenting an overview of relevant research studies that delve into collaboration within the field of software engineering.

The work by (Sun and Ryu, 2017) classified client-side JavaScript research into six research topics. The study aimed to help researchers and developers grasp the significant picture of JavaScript research more efficiently and accurately. While analyzing the evolution of a JavaScript application over time, (Chatzimparmpas et al., 2019) found that JavaScript applications undergo continuous evolution and expansion. However, the complexity persists consistently, and there is no discernible decline in quality over time. Study of (Jensen et al., 2009) presented a static analysis infrastructure for JavaScript programs using abstract interpretation in their work. While this early work was done in 2009, when developers lacked proper linters, modern-day tools provide a more accurate and precise analysis of code and its evolution. From the interview-based qualitative analysis by (Tómasdóttir et al., 2017), it can be inferred that developers utilize linters to prevent errors, sustain code uniformity, and expedite the process of code assessment. The study by (Ferenc et al., 2013) provides an overview of software product quality measurement, focusing on maintainability. It evaluates tools and models on numerous open-source Java applications. (Stamelos et al., 2002) stated a concern regarding the quality of the open-source code. According to the work, the open-source community should give considerable attention to enhancing the quality of their code. In order to ensure the industry standard, the open-source product has to be compared with the contemporary work of the software industry. Although many works related to quality analysis of open-source software have been done (Jarczyk et al., 2014) (Spinellis et al., 2009) (Adewumi et al., 2016), little has been done to analyze the open-source JavaScript projects and existing code evaluation tools.

In order to analyze the code quality, universally accepted metrics need to be used. Otherwise, one organization's quality standard will be different from others (Belachew et al., 2018). Several such standard metrics are mentioned by (Barkmann et al., 2009),

such as Coupling Between Weighted Method Count (WMC) using McCabe Cyclomatic Complexity and Lines Of Code (LOC). An empirical study by (Rahmani and Khazanchi, 2010) suggests a potential correlation between the number of developers and software size, offering insights into defect density. According to (Saboury et al., 2017), code smells negatively affect applications, and by removing code smells and issues, the application hazardous rate gets reduced by 65%. However, limited light has been shed on metrics like issue density, average estimated errors, lint errors, and cognitive complexity in open-source JavaScript projects.

3 METHODOLOGY

The research approach is depicted in Figure 1, presenting an overview of our methodology. Our study answers RQ1, RQ2, and RQ3 through a quantitative analysis using the project data we have collected. Code quality analysis is done based on 10 code quality metrics, such as maintainability, average estimated error, and lint error.

We investigate the 28 pre-existing tools for analyzing code quality metrics. Out of these, we selected six tools: Plato, JSHint, DeepSource, DeepScan, SonarCloud, and Codacy, from which we obtained the 10 metric values.

3.1 Project Selection

While selecting the project, diversity is maintained in terms of project size to generalize our study. The LOC, number of contributors, and commits in a project help formulate the size of that project. A total of 200 open-source JS projects are chosen from GitHub to conduct our analysis.

3.2 Developer Selection

A total of 380 developers' profiles have been gathered to analyze how code metrics vary depending on the level of developer expertise. The 380 developers are selected from a total of 4,056 profiles from the aforementioned 200 GitHub projects. The selection of 380 profiles from the overall group of developers is carried out arbitrarily.

3.3 Tool Selection and Analysis

In this study, initially, we examine a total of 28 online and offline tools capable of evaluating JS projects. Although most of the tools provide us with significant software metrics, not all of them are equally effective in metric analysis. Ultimately, performing a metric-based analysis on 26 features, we select the six most effective tools for our study - Plato [3], JSHint [4], DeepSource [5], SonarCloud [6], DeepScan [7] and Codacy [8]. These six tools offer a comprehensive range of metric reports, making them our choices to utilize in code quality analysis of JavaScript Projects for our research study.

The 26 features that formed the basis for our selection of 6 most effective tools are - 1) Anti-Pattern Issue, 2) Autofix, 3) Characteristics-based Rating, 4) Code Smell, 5) Cognitive Complexity, 6) Cyclomatic Complexity, 7) Dependencies, 8) Documentation Coverage, 9) Documentation Issue, 10) Duplication, 11) Estimated Error, 12) Function Analysis, 13) Function Count, 14) Graphical Report, 15) Total Number of Issues, 16) Lint Error, 17) Maintainability, 18) Performance Issue, 19) Project's Overall Rating, 20) Overall Account Overview, 21) Security Issue, 22) Style Issue, 23) Test Coverage, 24) Total Files, 25) Total LOC, 26) Variable Analysis.

A brief overview of the utility of these tools is presented herein.

Plato. Plato (Mousavi, 2017)[3] is one of the popular evaluation tools that analyzes projects and their files, detecting lint error, estimated error, complexity, and maintainability.

JSHint. This online tool[4] has variable-based and function-based analysis (Mousavi, 2017). However, manual code copying to the console for the report can be ineffective for big projects.

DeepSource. DeepSource[5] is an online static analysis tool (Higo et al., 2011) that establishes connectivity to the user's GitHub environment and carries out repository analysis upon activation.

DeepScan. The distinctive attribute of Deepscan (Alfadel et al., 2023)[7] is its capacity to modify analyzer engines. The tool can produce a report using its proprietary and ESLint engines simultaneously.

SonarCloud. SonarCloud (Raducu et al., 2020)[6], a static analyzing tool designed to assist developers in producing secure code.

Codacy. Codacy (Ardito et al., 2020)[8] provides the overall rating of the project. Furthermore, Codacy can also generate comprehensive reports on an entire GitHub account.

This study involves analyzing the relationship

[3] https://github.com/es-analysis/plato
[4] https://jshint.com
[5] https://deepsource.com
[6] https://www.sonarsource.com/products/sonarcloud/
[7] https://deepscan.io
[8] https://codacy.com

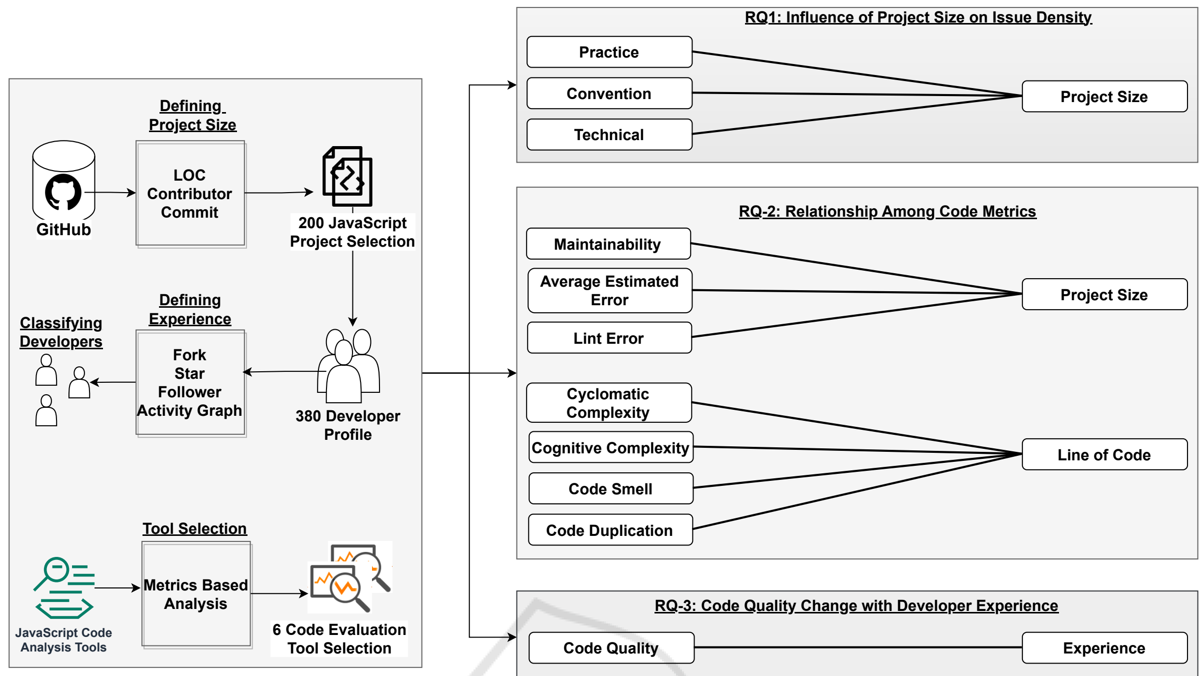

Figure 1: Overview of our research methodology and analysis.

among 10 code quality metrics, as detailed in the following sections. Table 1 presents the list of metrics employed in our study, as provided by the six selected tools.

Table 1: Available metrics from the used tools.

Metrics	Available from
Issues	SonarCloud, DeepScan, Codacy, DeepSource
Maintainability	Plato, SonarCloud
LOC	Plato, DeepSource, DeepScan, SonarCloud
Lint Error	Plato
Average Estimated Error	Plato
Cyclomatic Complexity	Plato, JSHint, SonarCloud
Cognitive Complexity	SonarCloud
Code Smell	SonarCloud
Code Duplication	SonarCloud, Codacy
Code Quality	DeepScan

3.4 Incorporating a Panel of Industry People

The study required the definition of project size and developer experience for analysis. A team of 25 software industry professionals, each with more than three years of experience in software companies, was incorporated to mitigate the possibility of bias. Their direct assistance in project and analyzer tool selection facilitated the creation of a neutral dataset for the study. Apart from this, the definition of metrics such as *"Project Size"* and *"Experience"* were obtained through an open-ended questionnaire administered to the panel of developers.

3.5 Research Questions

At the initiation of the work, we set forward three research questions before diving into the derivation of metrics and analysis. These questions guided us through the methodologies and analysis of the artifacts. Our intended three research questions, which are premised on section 1, are explained below.

RQ1: How Does Project Size Influence Issue Density?

The widespread use of JS has led to the development of many conventions among developers regarding issues, code structure, and patterns. Here, we aim to investigate the impact of project size on *Issue Density* (number of issues per 1,000 lines of code)[9] during the software development process.

[9] http://tinyurl.com/2s4y4nza

Here, we investigate code issues in JS projects and how they affect projects as the project size increases. The issues are classified into three broad subcategories. 1. **Practice** (issues regarding one's coding practices or different coding patterns among the members of a team), 2. **Convention** (Issues regarding the common practices that increase the code quality and maintainability), 3. **Technical** (issues in the remnant of codes or problematic codes that might create Runtime errors).

- ***Practice.*** For practice, problematic habits are considered as the assignment of the same name to different functions at different places, unused variables, use of different coding conventions at different parts of code, the existence of unreachable code, linting faults, same import statement at different places, etc.
- ***Convention.*** Convention criteria include anti-patterns, Null comparison, missing keys in item list complex code, big and bulky class files, etc.
- ***Technical.*** Lastly, in terms of technical aspect problems like remaining console logs from the development phase that can cause security issues, use of *'this'* statement outside of class, use of *'await'* functions inside of loops, declaration of variables after calling the variable, etc. has been considered.

The code issues may differ depending on the project size. To address the research question, we establish the definition of *Project Size*. The *Project Size* is subsequently compared to the *Issue Density*.

Defining 'Project Size'. To understand the distribution of issues, we measure it with respect to issue density. The issue density is compared with the change of project size. The project size is defined based on Line of Code ***'LOC'***, number of contributors ***'t'*** and number of commits ***'C'***. LOC is the direct and most widely used indicator of the size of a project (Barkmann et al., 2009). However, because of the limitation of LOC as a metric to measure project size, we incorporate the number of contributors and the number of commits along with it. Both the number of contributors and commits are reliable metrics for this purpose (Jarczyk et al., 2014).

Combining all these three metrics, the *"project size"* is defined as below -

$$p = (LOC \times 1) + (t \times 7) + (C \times 0.3)$$

Here, P = Project size
LOC = Line of Code
t = Number of contributors to the project
C = Number of commits.

The weighting factors of ***'LOC'*** is 1, ***'t'*** is 7 and ***'C'*** is 0.3 are considered based on their importance. The more the contributor in a project, the larger the project size grows. The values of ***LOC***, ***t***, and ***C*** are accounted for up to the time of our analysis.

In order to categorize projects according to their size, we have defined a project size classification scheme. The size value of each project is used to classify it into one of four distinct classes based on the project size classification scheme. This information is presented in Table 2.

Table 2: An overview of defining project size based on project scale value.

Project Size	Size Value (p)	Project Count
Small	<1000	50
Medium	1000 to 10,000	50
Large	10,000 to 30,000	50
Very Large	$>30{,}000$	50

In accordance with project size, a *Small* project is defined as having a size value less than 1000, while projects with a size value between 1000 and 10,000 are classified as *Medium*. Similarly, projects with a size value between 10,000 and 30,000 are classified as *Large*, and those with a size value greater than 30,000 are classified as *Very Large*. This classification scheme enables us to better understand and analyze project characteristics, as well as compare and contrast different projects based on their size. We took 50 projects from each of the classifications, which resulted in a total of 200 projects to avoid any kind of biases.

As the definition and the associated weighting factors are formulated in consultation with a panel of 25 experienced software industry professionals, it ensures that the weighting factors are informed by real-world expertise and are relevant to contemporary software development practices.

RQ2: What Are the Relations Among Different Code Quality Metrics, and How Do They Influence Each Other?

The question addresses how different software metrics are applied on JS code bases. We specifically focused on the relationship between project size, Line of Code (LOC), and metrics like maintainability, aver-

age estimated error, lint error, cyclomatic complexity, cognitive complexity, code smell, and code duplication. We break the research question into the following seven sub-questions based on code metrics.

- **RQ2.1.** What is the relationship between Project Size and Maintainability?
- **RQ2.2.** What is the relationship between Project Size and Average Estimated Errors?
- **RQ2.3.** What is the relationship between Project Size and Lint Errors?
- **RQ2.4.** What is the relationship between Total LOC and Cyclomatic Complexity?
- **RQ2.5.** What is the relationship between Total LOC and Cognitive Complexity?
- **RQ2.6.** What is the relationship between Total LOC and Code Smells?
- **RQ2.7.** What is the relationship between Total LOC and Duplication?

Table 3 demonstrates the tools selected for each metric examined in our study.

Table 3: Tools utilized for metric derivation.

Tool	Metrics derived from
DeepScan	Code Quality
DeepSource	Issues
SonarCloud	Code Smell, Cognitive Complexity
Plato	Maintainability, Lint Error, Avg Estimated Error, LOC
JSHint	Cyclomatic Complexity
Codacy	Duplication

The metrics used to answer the research question are briefly explained below -

Maintainability. Software maintainability refers to the degree of ease with which modifications can be made to a software system or component, including the correction of faults, enhancement of performance, or adaptation to a dynamic and evolving environment (159, 1990). The software maintainability is taken from *Plato*, which is calculated based on the Halstead Complexity Measures.

Average Estimated Error. The Standard Error of the Estimate (SEE) serves as a measure of the precision of predictions made regarding a particular variable. The square root of the mean squared deviation represents the value of SEE (Lederer and Prasad, 2000). The Average Estimated Error is taken from *Plato*, where the value is calculated using the Halstead-delivered bug score.

Lint Error. An error or warning message generated by a linting tool or piece of software that points out a potential issue or a breach of best practices in the code is known as a lint error (ans Kunst, 1988). It is calculated based on the problematic code patterns compared against predefined rules in ECMAScript with the help of *Plato*.

Cyclomatic Complexity. Cyclomatic complexity is a software metric that counts all potential routes through the code to determine how difficult a software system is (Samoladas et al., 2004). Cyclomatic complexity is derived from *JSHint*, which evaluates both the count of decision points and the nesting depth of control flow structures and ternary operators.

Cognitive Complexity. A software metric called cognitive complexity gauges how challenging it is to comprehend a codebase (Campbell, 2018). Cognitive complexity is taken from *SonarCloud* which considers nesting depth, specific constructs, and readability, to provide cognitive complexity.

Code Smell. It refers to code that might not be immediately evident to the developer but could lead to issues later due to potential issues, inefficiencies, or breaches of standard practices (Santos et al., 2018). Code Smell is identified by *SonarCloud*. It doesn't have a single formula for calculating code smells but leverages a combination of static code analysis, rule-based detection, and customizable settings to identify code smells.

Code Duplication. The act of repeatedly writing the same or similar code inside a codebase is referred to as code duplication (Rieger et al., 2004). *Codacy* detects clones or sequences of duplicate code present in at least two distinct locations within the source code of a repository.

RQ3: How Does Code Quality Change with Developer Experience?

It can be postulated that certain changes in the code quality exist with the progress of the developers' experience. In order to validate the intuition, the term *"developer experience"* is introduced as a potential quantitative measure consisting of four aspects: *forks, stars, followers,* and *activity graph.*

Defining 'Developer Experience'. Out of the aforementioned 200 projects, a selection was made of 380 developer accounts to compare their experience with code quality. The aspects considered while defining *developer experience* are elaborated below -

Fork. A fork is a copy of a repository a user or organization creates on GitHub to make changes or additions to the source. The total number of forks that are

present in a developer's entire repository base is considered. Fork count is a significant indicator (Jarczyk et al., 2014) of how many people use the developer's code as a foundation for their own projects.
Stars. Stars function as a "bookmark" of some sort in GitHub. Starring a repository is another way to express gratitude for someone, one finds helpful or instructive. A project is likely to have a lot of stars if it is of good quality (Jarczyk et al., 2014).
Followers. When a user is followed on GitHub, notifications about their activity on the site are sent to the follower. The number of followers is considered a reliable indicator of a developer's level of renown in the field (Blincoe et al., 2016).
Activity Graph. The graphical presentation, known as the activity graph, on GitHub showcases a user's contributions to a particular repository over a specified period. This particular visual representation is called the contribution graph or the GitHub heatmap. It is a helpful tool for assessing the activity levels of the selected 380 developers on GitHub. We incorporated a range from 1 to 10 score for the analysis.
Combining all these aspects, we formulate the *"developer experience"* as below -

$$developer - experience = k + s + (f \times 2) + g$$

Here, k = total number of forks in all the repositories of a developer
s = total number of stars in all the repositories of a developer
f = number of follower
g = activity graph value

The weighting factors of ***'k'*** is 1, ***'s'*** is 1, ***'f'*** is 2, and ***'g'*** is 1 are considered based on their importance. The number of followers (f) is considered a more vital factor in comparison to fork count, star count, and activity graph in a developer profile. Our analysis considers the values of $\boldsymbol{k}, \boldsymbol{s}, \boldsymbol{f}$ and $\boldsymbol{g}$ up to the present time of conducting this study.

Subsequently, the developer experience is compared with code quality to analyze if the quality differs with the difference of the developer experiences. A brief overview of code quality is expressed below -
Code Quality: The entire level of excellence and maintainability of software code is referred to as code quality. It may include a broad range of elements, like readability, scalability, robustness, security, maintainability, and many others, that support the efficiency and dependability of code. To assess "Code Quality," we employed the DeepScan code evaluation tool. DeepScan evaluates the readability, reusability, and refactorability of project code based on 61 predefined programming rules. Subsequently, it integrates the values of readability, reusability, and refactorability to generate a code quality score.

Table 4: An overview of developer experience with number of developers.

Experience Range	Experience Label	No. of Developers
<100	Novice	145
100 to 1000	Intermediate	118
>1000	Expert	117

A categorization of developer experience was established based on developers' experience value. Table 4 presents an overview of the ranges of developer experience along with the corresponding number of developers encapsulated within each range. Developers can be classified into three distinct levels of proficiency according to their experience level. In the realm of software development, individuals whose accumulated experience measures below 100 units are classified as Novices, whereas those whose experience ranges between 100 and 1000 units are designated as Intermediate. Ultimately, individuals with a developer background bearing an experience value surpassing 1000 are deemed proficient experts. This classification facilitates the analysis, compares the performance of developers at different skill levels, and identifies patterns and trends in their development activities.

This procedure entails the consultation of a panel comprising 25 industry experts to obtain their recommendations on the weighting factors and the criteria utilized for categorizing developers into the three delineations.

4 RESULTS AND ANALYSIS

We present the results of our research study in this section, which are organized according to each research question.

4.1 Result of RQ1: Influence of Project Size on Issue Density

We commence here by presenting the results for RQ1. Figure 2 represents the graphical depiction of the influence of project size on issue density. A noteworthy observation is the apparent decrease in issue density as the project size increases.

Interpreting the results for *practice* related issues, it is evident that the number increases from *Small* to *Medium* scale projects. This is anticipated because, for *Small* projects, beginners tend to depend on online resources and strictly follow documentation and other

artifacts. However, as they gain more experience and transition to *Medium* sized projects, they start to engage in more hands-on coding, incorporating a blend of various methodologies. With increasing experience and a shift towards *Large* scale projects, the coding practices become more streamlined, and issues regarding practices are reduced. Finally, for *Very Large* projects, there are generally several contributors. As different people come from different backgrounds and bring their practices to the mix, the issues seemingly increase here.

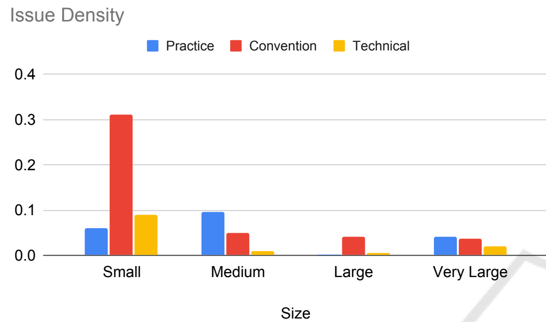

Figure 2: Influence of Project Size on Issue Density.

A discernible trend is evident for *convention* and *technical* issues as the projects grow larger and developers gain experience. However, when transitioning to *Very Large* projects from *Large* scale projects, opportunities for gaining additional experience become limited, causing more issues to arise in the project.

4.2 Result of RQ2: Relationships Among Code Metrics

4.2.1 Project Size and Maintainability

Figure 3 illustrates the influence of project size on maintainability. At the leftmost side of Figure 3, a noticeable incline indicates the high maintainability of small-scale projects. The underlying reason for this phenomenon is that *Small* scale projects generally lack the necessary amount of complexity to cause a decrease in maintainability, even with inexperienced developers. But as the scale of the project expands, there is a perceptible deterioration in its maintainability till 5000. As the project surpasses this threshold, it moves toward *Medium* scale projects that are mostly managed by more experienced developers. With the inclusion of their development experience, the code quality elevates, causing the maintainability to resurgence around the 7000 mark. Beyond the threshold of inflation, a gradual decrease in maintainability is observed. With the progression toward larger-scale projects, the inner complexity and failure points increase, which is counterbalanced to some degree by the developer's expertise. Moreover, coding conventions, standards, and documentation are maintained for *Large* projects. As such, a gradual and limited decrease in the maintainability of projects is observed, in contrast to a sudden and drastic decline.

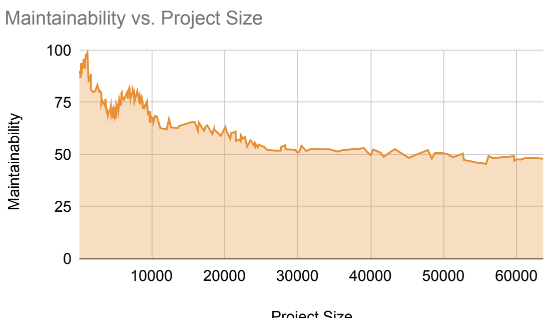

Figure 3: Project Size and Maintainability.

4.2.2 Project Size and Average Estimated Error

Figure 4 depicts a trend where the estimated error tends to grow as the size of the project grows. Although there are a couple of deviated spikes, the data mostly lead to the polynomial increase of average estimated error per file with the gradual expansion of project size. As the scope of a project grows, there is a corresponding increase in the line of code contained within individual classes or packages. An increase in the LOC presents a correlating rise in the likelihood of error. As a consequence, the mean estimated error tends to escalate.

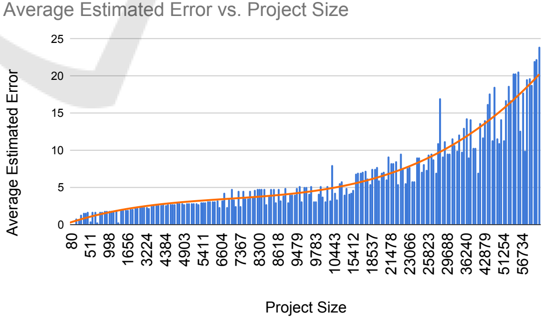

Figure 4: Project Size and Average Estimated Error.

4.2.3 Project Size and Lint Error

The present study illustrates the polynomial relationship between Project Size and Lint Errors, as depicted in Figure 5. Based on the analysis, there is a positive association between the increase in project size and the occurrence of lint errors. However, it contains a deflection of this general tendency, which is justifi-

able as the deviations are not a common scenario. So, with the increase in project size, the chances of lint errors in the project also escalate.

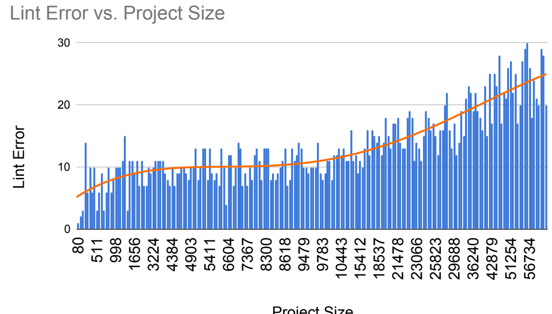

Figure 5: Project Size and Lint Error.

4.2.4 Total LOC and Cyclomatic Complexity

Figure 6 portrays the exponential relationship between Total LOC and Cyclomatic Complexity for an entire project. The aforementioned relation denotes a consistent plateau in the cyclomatic complexity up to a limit of 10,000 LOCs. Beyond this margin, there is an exponential rise of cyclomatic complexity with the increment of total LOC. It affirms that for *Large* and *Very Large* projects, the cyclomatic complexity displays an exponential growth pattern as the scope of the project advances.

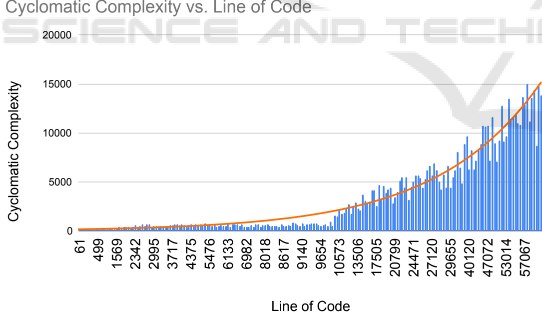

Figure 6: Total LOC and Cyclomatic Complexity.

4.2.5 Total LOC and Cognitive Complexity

The inter-relationship between Total LOC and Cognitive Complexity is illustrated in Figure 7. The outcome is a mirror of the relation between LOC and cyclomatic complexity. This is because cyclomatic complexity can be used as a factor while calculating cognitive complexity and the cognitive complexity increases proportionally with the cyclomatic complexity (Wijendra and Hewagamage, 2021).

There is a steady state with the increase of LOC until the LOC is 10,000. Beyond this point, the cognitive complexity grows exponentially with the growth of project size. The finding asserts a positive association between the expansion of project size and the augmentation of cognitive complexity, observed explicitly in large and very large-scale projects.

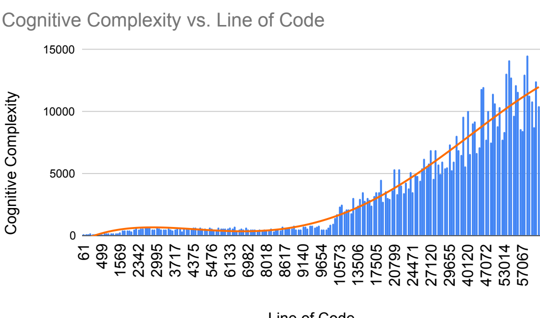

Figure 7: Total LOC and Cognitive Complexity.

4.2.6 Total LOC and Code Smell

The relationship in Figure 8 indicates the steady state of code smell until 10,000 with the increase of Line of Code (LOC). Later, there is an exponential rise of code smell with the expansion of total LOC. The above-mentioned finding corroborates the climbing nature of code smell with the enlargement of project LOC for *Large* and *Very Large* projects.

The graph representing cyclomatic complexity (see Figure 6) demonstrates a significant similarity, indicating a chance of the existence of a commensurate relationship between code smell and cyclomatic complexity. Although cyclomatic complexity can serve as a reliable indicator of potential code smells, other factors such as code duplication, inappropriate naming conventions, and the presence of large classes may also be responsible for code smells. While a relationship between code smells and cyclomatic complexity exists, both metrics provide unique insights into the code's quality and to use them in conjunction for a more comprehensive analysis opens up a future scope of study on this.

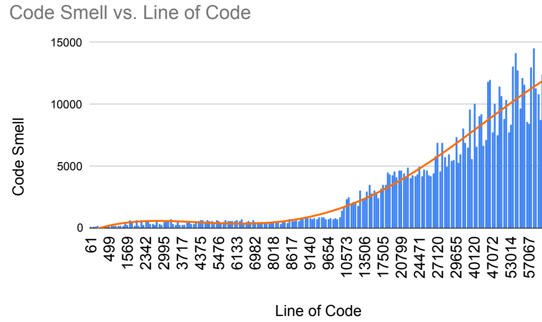

Figure 8: Total LOC and Code Smell.

4.2.7 Total LOC and Duplication

Figure 9 depicts a polynomial relationship between total LOC with code duplication. With the increase in the total LOC of the projects, the duplication tends to increase polynomically. The graph presented below indicates a proportional relationship between a project's total LOC and the occurrence of code duplication.

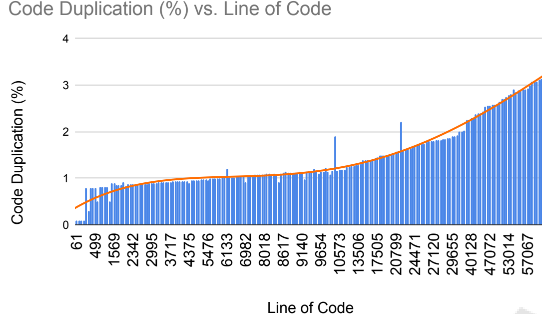

Figure 9: Total LOC and Duplication.

4.3 RQ3: Code Quality Change with Developer Experience

The diagram in Figure 10 depicts a clear correspondence between the code quality and the experience level. Based on our definition of *developer experience*, *Novice* developers (*Experience* < 100) with limited expertise typically oversee smaller projects characterized by lower code complexity, and as a result, their code quality remains high. As novice developers gain expertise, they often engage in larger projects, initially leading to a decline in code quality. Notably, with the increased experience, the code quality increases linearly, as expected for both the *Intermediate* ($100 <$ *Experience* < 1000) and *Expert* developers (*Experience* > 1000).

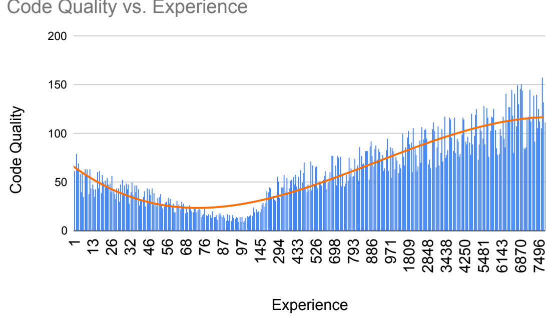

Figure 10: Experience and Code Quality.

Novice developers typically exhibit an average code quality ranging from 20 to 50, while intermediate developers tend to have quality values between 40 and 100. In contrast, experts typically demonstrate a code quality range of 100 to 150. The data demonstrates a positive relationship between developer experience and code quality on a general level.

5 THREATS TO VALIDITY

Regardless of our meticulous endeavors to collect and analyze data carefully, we recognize the presence of several limitations within our study.

5.1 Internal Validity

5.1.1 Outlier Data

Although caution was taken in selecting projects of a diverse range to represent the whole dataset, some outlier data still remained. However, the numbers are negligible to have any effect on the accuracy of the results.

5.1.2 Validation of Evaluation Tools

While the six tools employed in the study are widely recognized and reliable, there is still a possibility of encountering erroneous metric reports from any of the evaluation tools. To ensure the accuracy of the reports, we conducted a cross-matching of outcomes among the six tools.

5.1.3 Experience of Project Developers in JavaScript

While we analyzed the experience of 380 developer profiles, we cannot determine their specific experience in JavaScript. GitHub contributions are primarily oriented towards projects, not individual developer language expertise. This makes it challenging to pinpoint a developer's experience with the specific language of JavaScript.

5.2 External Validity

5.2.1 Defining Project Size and Developer Experience

"Project Size" and *"Developer Experience"* are the two terms defined for the analysis of this study. While an experienced developer team has validated the definition of *Project Size* and *Developer Experience*, all the developers are from Bangladesh. Thus, the validation and suggestions may reflect some specific regional perspective. The utilization of expert input dur-

ing this process enhanced the validity and reliability of the definition, thereby increasing researchers' trust in the outcomes of this study.

5.2.2 Considering Public Projects

This study utilizes the public repositories sourced from GitHub. The metrics of private repositories remain undisclosed because of their unavailability. This poses a challenge in asserting the generalizability of the study.

6 CONCLUSION

This paper presents an empirical study aiming at open-source JavaScript code quality analysis using code evaluation tools. We demonstrated that small projects start with high maintainability but decline as they grow. Beyond that, a shift to medium-scale projects managed by experienced developers leads to a resurgence, while larger projects experience a limited decrease. The increase in project size and experience caused issues to decrease in the project. However, while the scope of the project grows, the average estimated error, lint error, tend to escalate. An observable increase in cyclomatic complexity, cognitive complexity, code smell, and code duplication accompanies the rise in lines of code (LOC).

Our findings will help developers make better decisions regarding the relationship between project size, experience levels, and code metrics, promoting improved code practices for JavaScript-oriented software development. Additionally, by revealing the capabilities of evaluation tools, our study provides valuable insights for practitioners, selecting the most suitable tools for code evaluation practices and fostering continuous improvement in the software industry.

REFERENCES

(1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84.

Abdurakhimovich, U. A. (2023). The future of javascript: Emerging trends and technologies. *Formation of Psychology and Pedagogy as Interdisciplinary Sciences*, 2(21):12–14.

Adewumi, A., Misra, S., Omoregbe, N., Crawford, B., and Soto, R. (2016). A systematic literature review of open source software quality assessment models. *SpringerPlus*, 5(1):1–13.

Al Jobair., A., Mohammad., S., Maisha., Z. R., Mostafa., M. N., and Haque., M. J. I. (2022). An empirical study on neophytes of stack overflow: How welcoming the community is towards them. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 197–208. INSTICC, SciTePress.

Alfadel, M., Costa, D. E., Shihab, E., and Adams, B. (2023). On the discoverability of npm vulnerabilities in node. js projects. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–27.

ans Kunst, F. (1988). Lint, a c program checker.

Ardito, L., Coppola, R., Barbato, L., and Verga, D. (2020). A tool-based perspective on software code maintainability metrics: a systematic literature review. *Scientific Programming*, 2020:1–26.

Barkmann, H., Lincke, R., and Löwe, W. (2009). Quantitative evaluation of software quality metrics in open-source projects. In *2009 International Conference on Advanced Information Networking and Applications Workshops*, pages 1067–1072. IEEE.

Belachew, E. B., Gobena, F. A., and Nigatu, S. T. (2018). Analysis of software quality using software metrics. *International Journal of Computational Science & Application*, 8.

Blincoe, K., Sheoran, J., Goggins, S., Petakovic, E., and Damian, D. (2016). Understanding the popular users: Following, affiliation influence and leadership on github. *Information and Software Technology*, 70:30–39.

Borg, M., Tornhill, A., and Mones, E. (2023). U owns the code that changes and how marginal owners resolve issues slower in low-quality source code. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 368–377.

Brito, H., Gomes, A., Santos, Á., and Bernardino, J. (2018). Javascript in mobile applications: React native vs ionic vs nativescript vs native development. In *2018 13th Iberian conference on information systems and technologies (CISTI)*, pages 1–6. IEEE.

Campbell, G. A. (2018). Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58.

Chatzimparmpas, A., Bibi, S., Zozas, I., and Kerren, A. (2019). Analyzing the evolution of javascript applications. In *ENASE*, pages 359–366.

Chren, S., Macák, M., Rossi, B., and Buhnova, B. (2022). Evaluating code improvements in software quality course projects. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pages 160–169.

Ferenc, R., Hegedűs, P., and Gyimóthy, T. (2013). Software product quality models. In *Evolving software systems*, pages 65–100. Springer.

Higo, Y., Saitoh, A., Yamada, G., Miyake, T., Kusumoto, S., and Inoue, K. (2011). A pluggable tool for measuring software metrics from source code. In *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pages 3–12. IEEE.

Hussain, S., Chicoine, K., and Norris, B. (2021). Empirical investigation of code quality rule violations in hpc applications. In *Evaluation and Assessment in Software Engineering*, pages 402–411.

Jarczyk, O., Gruszka, B., Jaroszewicz, S., Bukowski, L., and Wierzbicki, A. (2014). Github projects. quality analysis of open-source software. In *Social Informatics: 6th International Conference, SocInfo 2014, Barcelona, Spain, November 11-13, 2014. Proceedings 6*, pages 80–94. Springer.

Jasim, M. (2017). *Building cross-platform desktop applications with electron*. Packt Publishing.

Jensen, S. H., Møller, A., and Thiemann, P. (2009). Type analysis for javascript. In *International Static Analysis Symposium*, pages 238–255. Springer.

Lederer, A. L. and Prasad, J. (2000). Software management and cost estimating error. *Journal of Systems and Software*, 50(1):33–42.

Molnar, A.-J. and Motogna, S. (2022). An exploration of technical debt over the lifetime of open-source software. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 292–314. Springer.

Mousavi, S. E. (2017). Maintainability evaluation of single page application frameworks : Angular2 vs. react.

Raducu, R., Esteban, G., Rodriguez Lera, F. J., and Fernández, C. (2020). Collecting vulnerable source code from open-source repositories for dataset generation. *Applied Sciences*, 10(4):1270.

Rahmani, C. and Khazanchi, D. (2010). A study on defect density of open source software. In *2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, pages 679–683. IEEE.

Rieger, M., Ducasse, S., and Lanza, M. (2004). Insights into system-wide code duplication. In *11th Working Conference on Reverse Engineering*, pages 100–109. IEEE.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. (2017). An empirical study of code smells in javascript projects. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 294–305. IEEE.

Samoladas, I., Stamelos, I., Angelis, L., and Oikonomou, A. (2004). Open source software development should strive for even greater code maintainability. *Communications of the ACM*, 47(10):83–87.

Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., Do Nascimento, R. S., Freitas, M. F., and De Mendonça, M. G. (2018). A systematic review on the code smell effect. *Journal of Systems and Software*, 144:450–477.

Spinellis, D., Gousios, G., Karakoidas, V., Louridas, P., Adams, P. J., Samoladas, I., and Stamelos, I. (2009). Evaluating the quality of open source software. *Electronic Notes in Theoretical Computer Science*, 233:5–28.

Stamelos, I., Angelis, L., Oikonomou, A., and Bleris, G. L. (2002). Code quality analysis in open source software development. *Information systems journal*, 12(1):43–60.

Sun, K. and Ryu, S. (2017). Analysis of javascript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)*, 50(4):1–34.

Tómasdóttir, K. F., Aniche, M., and van Deursen, A. (2017). Why and how javascript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 578–589. IEEE.

Wijendra, D. R. and Hewagamage, K. (2021). Analysis of cognitive complexity with cyclomatic complexity metric of software. *Int. J. Comput. Appl*, 174:14–19.

Zozas, I., Anagnostou, I., and Bibi, S. (2022). Identify javascript trends in crowdsourcing small tasks. In *International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 179–204. Springer.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS