The Adoption of JavaScript Linters in Practice: A Case Study on ESLint

Tómasdóttir, Kristín; Aniche, Maurício; van Deursen, Arie

DOI
10.1109/TSE.2018.2871058

Publication date
2020

Document Version
Accepted author manuscript

Published in
IEEE Transactions on Software Engineering

The Adoption of JavaScript Linters in Practice: A Case Study on ESLint

Kristín Fjóla Tómasdóttir, Maurício Aniche, Arie van Deursen
Delft University of Technology - The Netherlands
kristinfjolato@gmail.com, {m.f.aniche, arie.vandeursen}@tudelft.nl

Abstract—A linter is a static analysis tool that warns software developers about possible code errors or violations to coding standards. By using such a tool, errors can be surfaced early in the development process when they are cheaper to fix. For a linter to be successful, it is important to understand the needs and challenges of developers when using a linter.

In this paper, we examine developers' perceptions on JavaScript linters. We study why and how developers use linters along with the challenges they face while using such tools. For this purpose we perform a case study on ESLint, the most popular JavaScript linter. We collect data with three different methods where we interviewed 15 developers from well-known open source projects, analyzed over 9,500 ESLint configuration files, and surveyed 337 developers from the JavaScript community.

Our results provide practitioners with reasons for using linters in their JavaScript projects as well as several configuration strategies and their advantages. We also provide a list of linter rules that are often enabled and disabled, which can be interpreted as the most important rules to reason about when configuring linters. Finally, we propose several feature suggestions for tool makers and future work for researchers.

1 Introduction

An important part of software development is to maintain code by keeping it readable and defect free. This is where static analysis tools can step in: they automatically examine source code and look for defects or any issues related to best practices or code style. These tools aid in finding issues and refactoring opportunities early in the software development process, when they require less effort and are cheaper to fix [1], [2]. Due to their many benefits, static analysis tools have become commonly used in software development [3].

There is an abundance of available static analysis tools, ranging from academic research prototypes to tools widely used in industry. These tools vary in functionality, use diverse approaches for static analysis and can be used for different languages [4]. Some tools focus on coding styles, code smells or general maintainability issues, while others try to identify faults in code, perhaps examining specific types of defects such as related to security or concurrency [5], [6]. One type of a static analysis tool is a linter, which often uses a relatively simple analysis method to catch non-complex errors and violations of coding standards.

In fact, a good amount of research has already been conducted on general static analysis tools, including how developers use and perceive these tools [7], [8], [6] as well as how such tools are configured in the wild [9], [3]. Research already showed that using static analysis tools does not come without its challenges. They are known to produce a high number of warnings which includes many false positives [8], [10]. Moreover, some warnings need not to be relevant for all projects and can therefore be perceived as false positives when tools are not configured appropriately [11], [2], [6].

Most of the current research does not focus on JavaScript, an evergrowing language with a vibrant community. JavaScript has become a very popular programming language in the last years and in fact has been the most commonly used language on GitHub since 2013 [12]. It is known as the language of the web and has recently also become popular for server side development, serving as a general-purpose language. A notable characteristic of JavaScript is its dynamic nature, which is unlike other popular programming languages such as Java. For example, it allows for generating new code during program execution, dynamic typing, and use of undeclared variables.

Partly due to its dynamic features, JavaScript is considered an error-prone language [13]. For example, it can be easy to introduce unexpected program behavior with simple syntactic or spelling mistakes, which can go unnoticed for a long time [14], [15]. A linter can therefore be especially useful for JavaScript to detect these types of mistakes. Additionally, as JavaScript has become widespread, it becomes more important to have tool support that aids developers in keeping JavaScript code maintainable, secure, and correct. In recent years, linters have increasingly become commonly used tools for dynamic languages such as JavaScript [3]. We thus hypothesize that our current knowledge about how developers make use of static analysis tools may not directly apply to the JavaScript ecosystem.

This study therefore aims at complementing the existing body of knowledge by understanding why and how developers use static analysis tools in real world JavaScript software systems, and to see which challenges they face. Furthermore, linters need to be incorporated to the development process and configured appropriately for a project. This can be done in different ways and can be a demanding process when there are many rules to choose from. We investigate what methods developers use to configure linters and how they maintain those configurations.

We use a mixed methods research approach which involves collecting a combination of qualitative and quantitative data [16]. We choose ESLint [17] as a case study as it is currently the most popular JavaScript linter [18]. First, we apply a qualitative method, inspired by Grounded Theory [19], to conduct and analyze interviews with 15 developers from reputable open source software (OSS) projects. These developers were identified to be actively involved with enabling and configuring ESLint. Next we perform a quantitative analysis on the usage and configurations of ESLint in 9,548 JavaScript projects on GitHub. Finally, to challenge and generalize the previous two analyses, we survey 337 developers from the JavaScript community. We ask them about their experiences and perceptions with using linters, employing the previously acquired knowledge as input to a questionnaire.

This paper extends our previous work "Why and How JavaScript Developers Use Linters" that appeared at the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017 [20]. Our previous work included a qualitative analysis of interviews with JavaScript developers, which is now extended with an extensive analysis of linter configurations in OSS projects, along with a survey distributed in the JavaScript community.

The main contributions of this paper are:

- Perceptions on the usage of linters, after interviewing 15 developers that have actively used and configured a linter in reputable OSS projects.
- An extensive analysis of linter configurations in over 9,500 JavaScript projects on GitHub, shedding light on the most common linter configuration patterns.
- A study on the experiences and perceptions of 337 JavaScript developers on linters and specific ESLint rules, via a survey distributed in the JavaScript community.

2 Background: Linters for JavaScript

Well known and much researched static analysis tools include FindBugs [21], CheckStyle [22] and PMD [23]. These tools all have a different focus. For example, FindBugs detects numerous defects, such as infinite recursive loops and references to null pointers, CheckStyle is focused towards following coding standards, and PMD detects both code style violations and possible defects. JavaScript linters work in a similar fashion where the best known and most popular ones include ESLint, JSHint [24], JSCS[1] [25] and, the first linter created for JavaScript, JSLint [26].

ESLint is the newest of these and has gained much popularity in the last two years in the JavaScript community. ESLint was designed to be an extremely flexible linter that is both easily customizable and pluggable. ESLint provides 236 base rules[2], grouped in seven categories designed to help developers understand their purpose: *Possible Errors, Best Practices, Strict Mode, Variables, Node.js & CommonJS, Stylistic Issues*, and *ECMAScript 6*. In Table 1, we provide the description and the number of rules in each of these categories. Example rules include *no-eval* (*Possible Errors*), which disallows the use of the notorious eval function in JavaScript [27], and *indent* (*Stylistic Issues*), which enforces consistent use of indentation. The description of each rule and category can be found in the tool manual [28]. Developers are required to specify which of these 236 rules should be turned on, or instead use publicly available presets. A preset is a set of rules that are made available to the public, such as the ones from Airbnb [29], Standard [30], or even ESLint's recommended settings [28].

ESLint does not come with a configuration that is enabled by default, and is instead extremely customizable [31]. There are several different ways to configure rules for ESLint: 1) specifying individual rules in a configuration file, 2) specifying individual rules with comments in a file containing code, 3) using a preset, and 4) using a plugin (additional rules that can be created by any developer and plugged directly into ESLint).

When specifying an individual rule, it is possible to disable a rule (with the settings *off* or *0*) or to enable it, either as a warning (*warn* or *1*) or as an error (*error* or *2*). The rule is turned off when applying *off* and will have no effect whatsoever[3]. When a rule is set to *warn*, each instance of the rule being broken will appear in the output when running the linter. Lastly, *error* will have the same effect as *warn* except it gives an exit code of 1, meaning it can break the build when ESLint is a part of the build process.

Some rules have more detailed options that further specify how the rule is applied. As an example, the rule *indent* can have the setting *tab* to enforce the usage of tabs for indentation, or the setting *2* to enforce using two spaces. Even further customization is possible with this rule by enforcing explicit indentation settings for different statements and expressions, such as switch statements or variable declarations.

We chose ESLint as the linter to be analyzed in this study as it is the most commonly used linter in the JavaScript community, with over 72M downloads in npm (JSHint, the second most popular, has approximately 56.5M downloads[4]) [18]. Of all JavaScript linters, ESLint also has the most active community around it where it has the most contributors on GitHub, highest number of commits and frequent releases. In addition, it offers the greatest amount of functionality and flexibility out of all well known linters, thus not excluding nor focusing on any specific type of linting such as only analyzing styling issues or solely identifying possible errors.

3 Methodology

The *goal* of this study is to understand how developers use JavaScript linters in real world software systems as well as the challenges that they face. To that aim, we propose the following research questions:

- **RQ$_1$**: Why do JavaScript developers use linters?
- **RQ$_2$**: Which linter configuration strategies do developers adopt in their projects?
- **RQ$_3$**: What linter rules do developers commonly enable and disable?

1. JSCS is no longer supported and the maintainers have joined forces with ESLint as of April 2016.

2. As of release v3.13.0 in January 2017.

3. Disabling rules often occurs when projects make use of a predefined preset and developers want to disable some of its rules.

4. Measured in May 2017.

Category	Description	Available rules
Possible Errors	Possible syntax or logic errors in JavaScript code	31
Best Practices	Better ways of doing things to avoid various problems	69
Strict Mode	Strict mode directives	1
Variables	Rules that relate to variable declarations	12
Node.js and CommonJS	For code running in Node.js, or in browsers with CommonJS	10
Stylistic Issues	Stylistic guidelines where rules can be subjective	81
ECMAScript 6	Rules for new features of ES6 (ES2015)	32
Total		**236**

TABLE 1: ESLint rule categories with ordering and descriptions from the ESLint documentation [28]

- **RQ$_4$**: What are the challenges in using a JavaScript linter?

To answer these research questions, we perform three different steps that involve three different sources of information: first, we interview developers with the goal of understanding why and how they use linters as well as the challenges they face; next, we mine existing JavaScript open source repositories to analyze their linter configurations; finally, to generalize the results obtained in the interviews, and to further explain what we observed in GitHub repositories, we survey developers from different JavaScript communities. Each step is described in the following subsections.

3.1 Part I. Interviewing JavaScript Developers

To answer RQs 1, 2, and 4, we followed a qualitative research approach in our study [16], inspired by many concepts of classic Grounded Theory [19], [32] where the aim is to discover new ideas emerging from data instead of testing any preconceived hypotheses. We also followed Stol et al.'s guidelines [33] that were derived after a systematic literature review on the usage of Grounded Theory in Software Engineering.

With an open mind we wanted to understand how and why developers use a linter for JavaScript. For that purpose we collected data by conducting 15 interviews with developers from reputable JavaScript projects on GitHub. We explain the process of conducting these interviews in Section 3.1.1. The interview recordings were manually transcribed and analyzed with continuous *memoing* and *coding*, which is further described in Section 3.1.2. Finally, we detail our participants in Section 3.1.3.

3.1.1 Interview Procedure and Design

The interviews were conducted in a semi-structured fashion as it is commonly done in software engineering research [34]. With this method, specific questions are combined with open-ended questions to also allow for unexpected information in responses. Hove and Anda [34] encourage interviewers to talk freely, to ask relevant and insightful questions and to follow up and explore interesting topics. These guidelines were followed while performing the interviews. Each interview was built upon a list of 13 standard questions.

To begin with, participants were asked broad questions which often provided an opportunity for further discussion. Example questions include: *Why do you use a linter in your project?* and *How do you create your configuration file and maintain it?*. Other questions were more specific, such as: *Do you experience false positives? if so, which?*. The complete list of questions is available in the paper appendix and in the extended online version [35].

Interviewees were asked to participate in an online video call. The interviews were recorded with permission and lasted from 16 to 60 minutes, with an average duration of 35 minutes. Three out of the 15 participants were not able to participate in an online call and instead received a list of questions via e-mail and provided written responses.

3.1.2 Analysis

Continuously after each interview, *memoing* was conducted to note down ideas and to identify possible categories in the data. The interview recordings were then ultimately manually transcribed. First, we performed *open coding* where the transcripts were broken up into related sentences and grouped together into the three main topics that drove our interviews (why and how developers use linters and the challenges they face). Secondly, we performed *selective coding* where more detailed categories were identified which became the topics we present in the Results (Section 4). In this process we took advantage of the memos that had been written over the course of conducting the interviews. The complete list of codes can be found in the appendix [35].

3.1.3 Interview Participants

In order to find potential participants for the interviews we examined the most popular JavaScript projects on GitHub, according to their number of stars in December, 2016. We conjecture that by observing the top projects on GitHub we can obtain an insight into active and reputable projects with many contributors, providing more interesting and relevant information about the usage of linters. We detected projects that 1) use ESLint, 2) have some custom configurations (*e.g.*, not only using a single preset) and 3) where one or two contributors could be identified that have been more involved than others in editing the configuration file. We then sent an e-mail to one or two main contributors of the ESLint configuration file of the corresponding project, briefly explaining the purpose of the study and asking for a short interview. These requests were sent out in batches of 5-10 e-mails (starting with the most popular projects) as it was difficult to predict how many positive replies we would receive. Batches were sent out until we had received a sufficient number of positive replies back, where the goal was to perform at least 10 interviews, or until we were satisfied with the amount of information we had collected.

TABLE 2: All participants' codenames, number of months using ESLint in the corresponding OSS project and the range for the project placement in the top 120 JavaScript projects on GitHub

Code	Months	Placement
P1	25	11-20
P2	22	11-20
P3	5	21-30
P4	14	21-30
P5	8	31-40
P6	7	41-50
P7	1	61-70
P8	23	71-80
P9	5	81-90
P10	3	81-90
P11	4	91-100
P12	16	91-100
P13	15	111-120
P14	24	111-120
P15	22	111-120

TABLE 3: Experience of participants, showing the lowest and highest answers along with the average of all answers

	Low	High	Average
Years as developer	3.5	27	11.8
Years as JavaScript developer	1.3	20	8.9
Years in project	0.6	5.0	2.7
Project age	1.0	8.0	5.1

A number of 120 projects were eventually examined where 37 requests were sent out. These resulted in 15 interviews being performed, thus with a response rate of 40%. The information from these 15 interviews was considered enough to provide us with *theoretical saturation* [19]. Table 2 shows the developers who participated in the interviews where, in order to keep the participants' anonymity, they are given names starting with the letter P and a number from 1 to 15. The months each corresponding project had used ESLint is also displayed[5], where most projects had migrated from another linter such as JSHint. The table also shows the placement of the projects in the top 120 JavaScript projects on GitHub within a range of 10 projects each (to maintain the participants' anonymity). Participants are ordered by the projects' number of stars on GitHub, and not by the order we interviewed the developers. A summary of the participants' experience is shown in Table 3 where the average experience as a professional software developer was 11.8 years. Among the 15 participants, four are founders of the project, seven identified themselves as lead or core developers and four are project maintainers.

3.2 Part II. Mining Linter Configurations in Open Source Systems

To answer RQ_3, we performed a quantitative analysis to know exactly how developers configure their linters and what the most common configuration patterns are. For this purpose, we analyzed 9,548 ESLint configuration files extracted from 86,366 JavaScript projects on GitHub. We analyzed how much configurations are applied by developers, whether they rely more on pre-made settings (presets) or their own configurations, and which types of rules are most commonly used.

3.2.1 Data Collection

To collect projects we chose GitHub as a data source due to the high number of available JavaScript projects and the convenience of retrieving the data [36].

The original data selection consists of all JavaScript projects on GitHub that have at least 10 stars and are not forks of other projects. The purpose of giving a star to a project on GitHub is to keep track of projects that a user finds interesting or simply to show appreciation to a repository [37]. By only including repositories with at least 10 stars the intent is to analyze "real" software projects. Kalliamvakou *et al.* [36] showed that a large portion of GitHub repositories are not for software development but for other functions such as for experimental or storage purposes. It is expected that repositories that were created for experimentation or testing purposes only, or pet projects that were started and abandoned, will not receive 10 stars from other users. Furthermore, forks of other projects were excluded to avoid having duplicate configuration files in the dataset. This resulted in 86,366 projects being collected to analyze.

To retrieve data on these projects, we used Google BigQuery [38] on the most recent GHTorrent [39], [40] dataset on GitHub projects from April 1st, 2017. The precise SQL query that was used to obtain the data can be found in our online appendix [35].

After retrieving the 86,366 project entries, additional filtering was performed on the dataset. First of all, there were examples of duplicate entries in the dataset where they point to the same GitHub API URL for the project. This can happen when the project name has been modified or when the owner has been changed for a repository, in which case a new entry is created in the GHTorrent dataset. The duplicate entry that had a more recent date for its last commit was kept in the analysis. This filtering resulted in 1,596 projects being removed from the dataset.

Secondly, even though the query includes a statement to not include deleted projects, some repositories could not be accessed. In seven cases, an HTTP error status code of 451 (unavailable for legal reasons) was returned when trying to access the repository. More commonly, or in 871 cases, the repository's URL could not be found, returning an HTTP status code of 404. Due to this filtering, a number of 878 additional projects could not be analyzed, resulting in a final number of 83,892 possible projects.

Besides removing forked, duplicated and deleted projects, no additional filters were applied such as regarding project size or activity. We decided not to filter the projects by a minimum size, as the intention was to analyze all different types of JavaScript projects: big or small, collaborative or personal. However it could therefore be the case that the resulting dataset includes projects that are not suited to ever use a linter, *e.g.*, a repository that is not a software but simply a collection of scripts or even tips for developers. Additionally, these could be projects that have not been active for several years, and perhaps even not active since ESLint was created in June 2013.

5. As of February 2017.

3.2.2 Extracting the Linter Configuration

While we only analyze the linter configurations for ESLint, we note down the usage of other linters as well, namely: JSHint, JSCS and Standard. For each linter the tool searches for a configuration file with a specific known name and file ending. The configuration file is typically located in the main directory of a project (as it will then be used for the whole project), so in order to save execution time and to simplify the tool, it is the only location where the tool searches for the file.

If a configuration file (either *.eslintrc* or *package.json*) is found for ESLint then it is retrieved. For the other linters the tool merely notes down their presence in order to measure their prevalence.

With the configuration file in hand, our tool extracts all the relevant information about enabled and disabled rules. We make the tool available for download in our appendix [35].

3.2.3 Dataset Characteristics

The sizes of the projects in GHTorrent are expressed in kilobytes (KB) where the first, second and third quartiles are the following: 126 KB ; 364 KB ; 1,928 KB. Moreover, 55,9% of the projects are 500 KB or smaller.

For the last commit date of the projects, 7.4% of the projects had their last commit before ESLint was first released (v0.0.2) in June 2013 and 35.4% of the projects before the first major release (v1.0.0) in July 2015[6]. Thus when analyzing the prevalence of linter usage in JavaScript projects or even more specifically, ESLint usage, some projects in the dataset are less likely to have used a linter. Moreover, when manually inspecting the dataset, we found examples of programming guides (example at [43]) and tutorials (example at [44]). Nevertheless, these types of projects were not filtered out as we wanted to avoid making any assumptions about the dataset and for it to be as broad and general as possible.

Table 4 shows the estimation of the usage of four linters in our dataset, where ESLint was the most commonly used linter followed by JSHint. We observe a similar pattern for npm, where ESLint is also the most downloaded linter, followed by JSHint, but with a greater difference [18]. In total 20,292 out of the 83,892 of the analyzed projects, or 24.2%, used a linter. We also show the number of projects that use more than one linter in Table 5. Only 9.1% of the projects that use a linter, use two or more linters, where 4.3% use ESLint and another linter. The percentage of projects using any of these linters appears to be higher for those that have more stars. Table 6 shows the number of projects that use any of these linters while observing only the top starred projects, such as the top 10 or top 1,000 projects. The percentage of projects using a linter steadily decreases when observing more projects, going from 70.0% for the top 10 projects to 28.5% for the top 30,000 projects.

3.3 Part III. Surveying Developers

To generalize the findings we obtained from the interviews and to further explain the results we obtained after mining configuration files on GitHub, we surveyed 337 developers from the JavaScript community about their perceptions and experiences with using linters. The survey was built upon the previously acquired knowledge where the information was used as input to both open and closed questions.

6. The exact dates that were used for this comparison were 08:00 June 30th 2013 [41] and 08:00 July 15th 2015 [42].

Linter	Projects with linter	% of all projects
ESLint	9,548	11.4%
JSHint	9,447	11.3%
Standard	1,651	2.0%
JSCS	1,578	1.9%
Total	**20,292**	**24.2%**

TABLE 4: Estimation of the number of projects using ESLint, JSHint, Standard and JSCS in our dataset

Linters used	Number of projects	% of all projects
1	18,450	22.0%
2	1,753	2.1%
3	88	0.1%
4	1	0.0%
Total	**20,292**	**24.2%**

TABLE 5: Estimation of the number of projects using multiple linters in our dataset

3.3.1 Survey Design

This survey was constructed under guidelines from both social sciences and software engineering research, *e.g.*, from Fink [45], De Vaus [46] and Kitchenham [47]. A shortened version of the survey is present in the paper appendix and a full PDF version is available online [35].

We mainly took advantage of closed questions to make the survey more reliable and more compelling for participants to complete. To minimize the risk of forcing opinions on participants, each closed question, where possible, contained an option where the user could write his or her own response, along with having a neutral option in ordinal questions. Moreover, to minimize the risk of bias due to the order of options in the survey, the options were randomly shuffled for each participant, wherever possible. To choose the options for the closed questions, we used input from the results of the two previous data sources.

Early on in the survey, participants were asked if they had ever used a linter in a project. If the answer to that question was negative, they were only presented with two

Top projects	Projects with linter	% of top projects
10	7	70.0%
100	65	65.0%
300	185	61.7%
1,000	535	53.5%
3,000	1,371	45.7%
5,000	2,082	41.6%
10,000	3,675	36.8%
20,000	6,306	31.5%
30,000	8,554	28.5%
Total	**20,292**	**24.2%**

TABLE 6: Estimation of the number of projects using a linter in our dataset

additional questions, asking how important they consider a linter and why they have never used a linter. Other participants received questions on the usage of linters, including a question asking which linters they had used. If a participant had not used ESLint in any project, he or she was not presented with questions that had specifically to do with ESLint rules. Instead they (and also those that had used ESLint) received more generic questions about why they use a linter, what methods they use to configure it and which challenges they have faced.

One of the main goals of the survey is to collect the perceptions of participants on the importance of individual ESLint rules. For this we derive a set of 14 rules in the four most important ESLint categories, as identified by the interviewees and in project configuration files, except for the *Variables* category where there are only 12 available rules. These sets were created with the following criteria: First, we select the five most commonly enabled rules where a preset is not used (according to our previous analysis) and five where a preset is also used. Some rules can appear in both these sets where additional rules are then chosen. If an even number of additional rules were needed, one rule was added from each list of commonly enabled rules (*e.g.*, the sixth top enabled rule without a preset and also the sixth top enabled rule with using a preset). If an odd number of additional rules were needed, the last chosen rule is the next rule of either set that has been more commonly enabled. The same process was applied to select the remaining four rules, except this time observing the most commonly disabled rules, consisting of two rules where a preset is used and two where no preset is used. Like with many other questions, the order of the rules was randomized, along with the order of the questions themselves as they appeared for each of the four categories.

Before publishing the survey we performed pilot studies, but rather in the nature of moderating focus groups. More specifically, six participants with different characteristics were recruited to take the survey in individual sessions with the first author. The pilot participants were colleagues of the authors that were known to have used JavaScript, with different levels of experience. The participants were asked to read each question out loud and to express their thoughts and understanding of the text. They were especially encouraged to speak out if they thought any questions were unclear or vague and to express whether answer options were appropriate. Furthermore, before answering the questions, the pilot participants were asked to read the introductory text to evaluate how appropriate and motivating it was. At the end, we discarded the data generated in the pilot studies.

3.3.2 Target Population

The survey was specifically directed towards JavaScript developers that have used a linter, and in particular those that have used ESLint, as they can answer more of the survey's questions and have more knowledge on the topic. We applied convenience sampling where the survey was advertised in several places on the web where it was likely to find members of the target population. More specifically, the survey was distributed in the following four locations (the references point to the direct post where the survey was promoted):

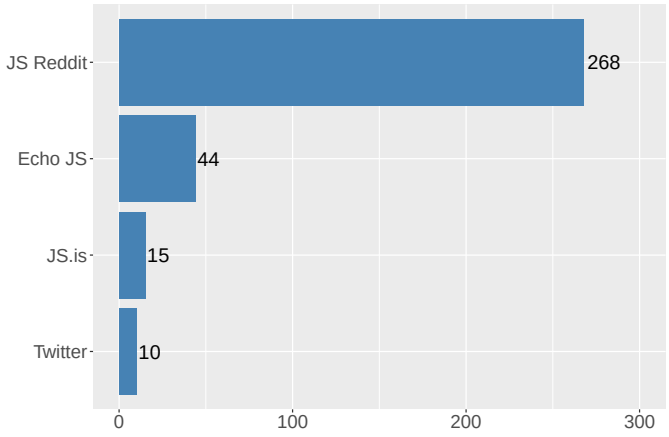


Fig. 1: Number of full responses from each location where the survey was distributed.

1) **JS.is**. An Icelandic JavaScript user group on Facebook with 789 members [48][7].
2) **JS Reddit**. A "subreddit" about JavaScript on the popular community-driven social news site reddit.com with 111,649 subscribers [49].
3) **Echo JS**. A community-driven news site about JavaScript development [50][8].
4) **Twitter**. A news and social networking site [51].

Distributing the survey in these four locations resulted in a total number of 337 completed responses, as shown in Figure 1. Furthermore, additional 476 partial responses were received, which we did not include in the analysis. The completion rate is therefore 42.0%. The survey was first posted on the web on June 1st and eventually closed on June 14th, thus being available for 14 days.

3.3.3 Analysis of the Survey Results

We apply descriptive statistics to report the findings of the closed questions. For the open questions, further manual analysis was needed. There are eight questions in the survey that are completely open (essay questions) but many more that are closed but additionally contain an open answer possibility to add an extra option. Similarly as when processing the interviews, an inductive method [45] was used where the main themes of each question were identified as they emerged when processing the answers. Each identified category received a code and each answer was labeled with at least one code. When needed, the coding was conducted on different levels of detail, where broad categories were identified which were then sub-classed into more detailed ones [46]. In cases where answers described more than one item, *e.g.*, listing several different reasons for something, it was decided to give multiple codes instead of choosing only one and then possibly disregarding parts of an answer. The derived analysis can be found in our online appendix [35].

7. All information about the four resources was collected on June 14th 2017.

8. This reference does not contain a direct link to the promotion post as it is not made accessible by the Echo JS website.

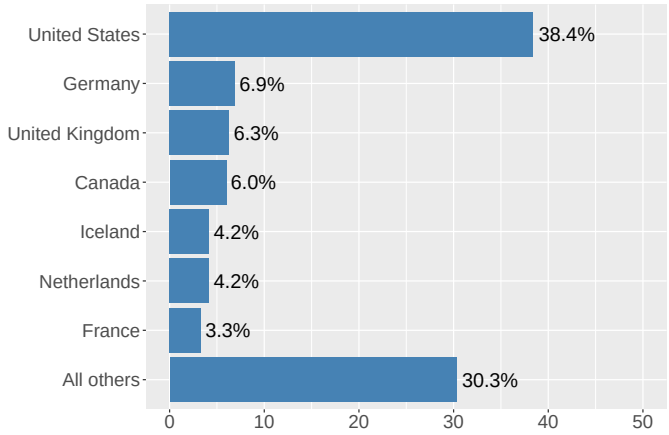


Fig. 2: The country of residence of participants

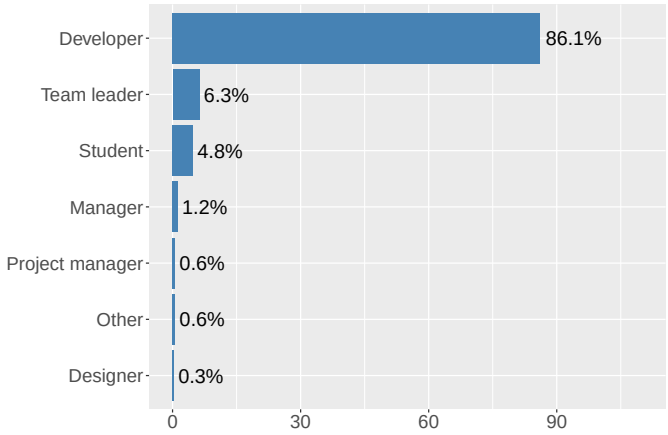


Fig. 3: The primary roles of participants in software development

3.3.4 Participants' Characterization

Nearly all of the survey's participants are male, or 96.4%, with only four female participants, accounting for 1.2%. One participant chose Gender Variant/Non-conforming while the rest preferred not to answer the question. The respondents' country of residence covers 53 different countries where more than half of all participants, or 69.7%, come from seven dominating countries as shown in Figure 2, most commonly from the United States.

A large majority of the participants identified their primary role in software development as a developer, or 86.1%, as shown in Figure 3. Other noticeable roles are team leader (6.3%) and student (4.8%).

The experience of the participants in software development and with JavaScript is shown in Figure 4. The average experience of the participants in software development is 8.0 years, and their average experience in working with JavaScript is 5.8 years. All participants claimed to have used JavaScript in the last year and 96.1% reported that it was one of their main programming languages.

The majority regularly works with commercial software (87.0%) while around half of the participants work with open source software (50.6%) (some participants chose both options). Finally, 93.7% claimed to have used a linter in a JavaScript project. We thus consider that our survey achieved our targeted population. The 21 participants that had never used a linter were not presented with the greater part of the rest of the survey.

4 Results

In the following we present our results on why and how JavaScript developers use linters, along with the challenges that they face.

4.1 RQ_1. Why Do JavaScript Developers Use Linters?

We first introduce the six reasons to why JavaScript developers use linters, which we derived from the 15 interviews. Each of the following sub-sections represent the reasons that emerged from our qualitative analysis. In addition, in Figure 5, we present the survey participants' agreements with the proposed reasons aforementioned[9]. Sections are ordered by their importance, according to the survey participants.

4.1.1 Maintain Code Consistency

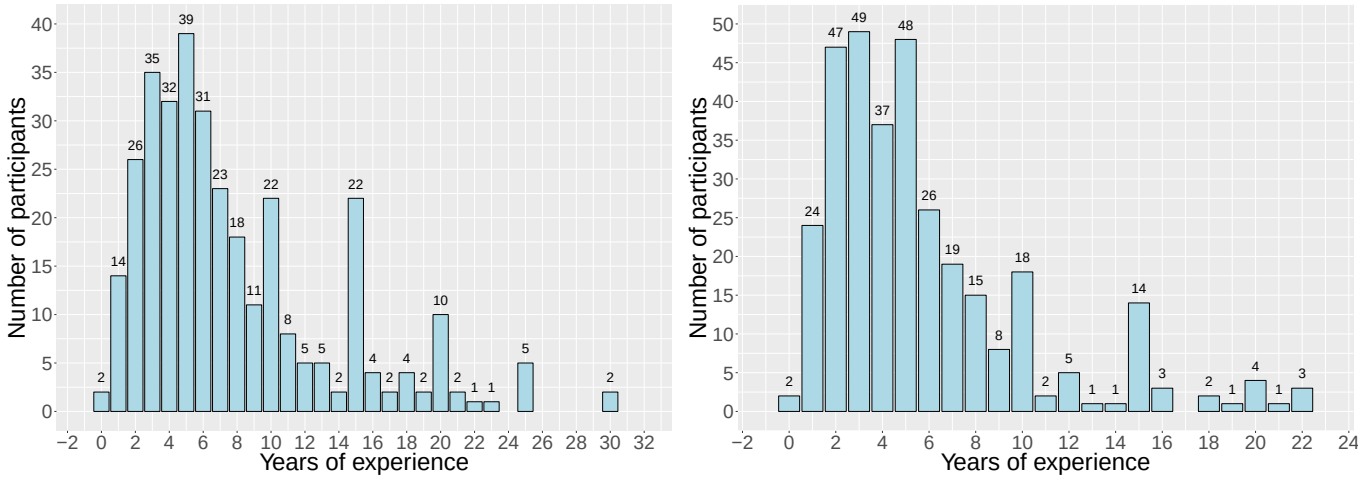
Every single interview participant mentioned that one of the reasons why they use a linter is to maintain code consistency. Moreover, 97% of our survey respondents also agree or strongly agree, and this was in fact the most agreed topic in the entire survey (Figure 5, Maintain code consistency).

Having a consistent code style in a project is beneficial for many reasons, one being that it improves the readability and understandability of the code. As an example, P10 reported that inconsistent code, such as having different spaces and semi colons, makes the code very difficult to read and understand since in those cases these inconsistencies consume all his attention. This might even be especially relevant in the case of JavaScript since it is a language where the developer has substantial freedom in how to write the code (P12, P14): "*With JavaScript you can write code in many ways, and it can be hard to read other people's code if you write it in a different way.*" (P12).

This topic relates mostly to the category *Stylistic Issues* where there are many different rules available to enforce specific code styles. Even though every participant mentioned this matter in the interviews, it does not seem to be of high priority for them. When participants were asked which category of rules they thought were the most and least important ones, two considered *Stylistic Issues* to be the most important category while 10 thought it to be the least important one.

Some participants were bothered by the fact that choosing which style to follow is a very subjective decision and developers generally have very different opinions on how code should be written (P1, P3, P5): "*Stylistic Issues - they're all opinion based.*" (P5). On the other hand, four participants explained that *Stylistic Issues* was indeed the least important category simply because other categories can catch bugs which is far more important (P2, P4, P9, P13). This category

9. The reasons presented in Figure 5 have been shortened to fit in the figure, and two of those reasons have been combined into one section in the text (namely, Automate Code Review and Avoid Giving Negative Comments).



(a) Experience in software development

(b) Experience with using JavaScript

Fig. 4: Participants' experience in software development and with JavaScript. Axes show years of experience and number of participants with the corresponding answer.

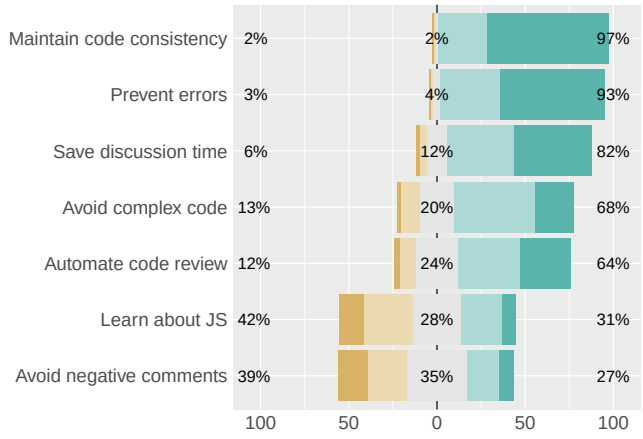


Fig. 5: Level of agreement with reasons as to why the participants have used a linter. Bars show Strongly Disagree, Disagree, Neutral/Not Applicable, Agree, and Strongly Agree, respectively. ($N = 309$)

thus still provides a lot of value and they would not want to omit it: "*They make [the code] harder to read but they just don't cause issues as much.*" (P13).

4.1.2 Prevent Errors and Augment Test Suites

Using a dynamic language such as JavaScript is not free of risks: "*Without a linter [JavaScript] is a very dangerous language. It's very easy to make a very big problem and then spend 30 minutes to find it.*" (P7). The majority of the interview participants reported that the number one reason as to why they use a linter is to catch possible errors in their code: "*There are things which are easy mistakes to make and are obvious errors and I think those provide the highest value. Because you have a one to one correspondence between times that a rule catches something and bugs that you've prevented.*" (P4). This is also inline with the results of our survey, as 93% of participants agree or strongly agree that this is an important category of issues (Figure 5, Prevent errors).

More explicitly, when asked about the most important category of warnings in ESLint, 10 interviewees answered that *Possible Errors* was the most important one (P1, P2, P3, P4, P5, P7, P8, P9, P11, P12): "*Possible Errors is the #1 most important, the biggest reason to use a linter is to catch errors the programmer missed, before they become a runtime bug.*" (P9). These rules can be especially useful for bugs that are hard to find and to debug (P15).

A special category of bugs in JavaScript has to do with the declaration of variables because of the dynamic nature of the language. Two of our interview participants reported that *Variables* was the most important category (P10, P15). When a developer *e.g.*, mistypes a variable or uses the wrong variable name, the linter can catch it and warn the developer: "*It's very easy to write JavaScript code that has errors, you might use a variable that hasn't been declared or you might have a typo in your variable name and because JavaScript is often not compiled, you'll only discover that much later when you run the code.*" (P1).

Nine participants (P1, P3, P4, P5, P8, P10, P12, P13, P15) mentioned the importance of the rule *no-unused-vars* (identifies variables that have been declared but never used) and five (P1, P3, P5, P13, P15) mentioned *no-undef* (identifies variables that have not been defined) which are both useful to identify mistyped variables. In addition, *no-dupe-keys* and *no-dupe-args*, two duplication rules for keys in objects and names in function parameters, were also mentioned by P3, P4, P8, and P15.

While linters are being used to catch errors in code, there is another popular and widely accepted method to catch bugs which is to write unit tests. It is therefore interesting to know how these two methods are combined for this purpose. Some participants mentioned that they use a linter on top of unit testing as a complementary approach to the regular tests (P1, P3, P8). P1 and P8 pointed out that unit tests commonly do not cover all code, which can result in problems being easily missed: "*You need to seek all possible cases for unit tests, but sometimes it's very hard, and of course in all projects, unit tests don't cover all possible cases. So this is why*

a linter is a second protection line." (P8). Furthermore, the tests can also take substantial time to run and thus the linter can be seen as a much faster version of smaller subtests (P1).

On the other hand, participant P4 believes that unit tests and manual tests can usually cover all errors, so even though ESLint would not be used, the errors would eventually be caught by the various tests that are applied. However, P4 says that the linter can catch them earlier in the process and is also better at identifying code that is ambiguous.

4.1.3 Save Discussion Time

Having a set of rules regarding code style can also save time that is spent on discussing different styles and opinions (P2, P4, P5, P6, P7, P10). 82% of our survey participants also agree or strongly agree (Figure 5, Save time on discussions).

In big projects with many contributors there can be many pull requests in circuit and discussions can occur where developers disagree on a certain style that is used. P2 explains that discussing code styles is not worth the effort when there are other more important things to discuss. He further describes that comments regarding code style on pull requests can be different depending on which developer is conducting the review. In some cases, contributors can therefore receive contradictory advice if no rules exist that everyone goes by.

The discussions about code style that can occur in pull requests or in issues can also even lead to arguments between people since developers have very different opinions on the matter (P1, P2, P3, P5). All this can be avoided by deciding upon a set of rules to begin with: "*It's almost like a code contract. There may be things that each of you have assumed and you don't know what your own assumptions are, and what could possibly lead to conflict down the road, so you have a written contract to try to address everything up front.*" (P7).

4.1.4 Avoid Ambiguous or Complex Code

It can be difficult to understand code correctly where the intention is not perfectly clear. The category *Best Practices* tries to tackle this problem where, according to its documentation [17], it contains rules that relate to better ways of doing things to help developers avoid problems. While only one participant recognized this category to be the most important one (P6), others identified it as the second most important (P4, P8, P13, P15). In our survey, 68% of participants agree or strongly agree that avoiding ambiguous and complex code is important, while 20% were neutral, and 13% disagree or strongly disagree (Figure 5, Avoid complex code).

Some of these rules try to prevent code from being misunderstood: "*It helps enforce code which says what it does, so that it's easy to understand.*" (P4). In some cases code is actually doing something else than it appears to and a linter can help to detect these situations (P2, P4, P6). For exemplary beneficial rules, participants mentioned restricting the usage of switch statements by forbidding the use of "fall throughs"[10] (P4), and disallowing unreachable code (P2, P3, P6, P15).

10. A "fall through" occurs when all statements after a matching case are executed until a break statement is reached, regardless if the subsequent cases match the expression or not.

4.1.5 Automate Code Review

Several participants mentioned that they use the linter to avoid having to manually review the code style in pull requests (P1, P2, P3, P4, P8, P11, P14, P15). Furthermore, it saves time for the contributor of the pull request since he or she receives much faster feedback from a linter than from a person that would conduct a review (P4). In the survey, we see that 64% of participants agree or strongly agree with the usage of linters for improving code reviews. However, only 12% disagree or strongly disagree with it (Figure 5, Automate code review).

Maintaining code consistency with a linter can also make pull requests much easier to review. When there is a set of stylistic rules in a project to which everyone has to conform, all pull requests have minimal stylistic changes. If there are no rules, there can be multiple code changes of *e.g.*, only whitespaces or line formatting which might be caused by different editors being used. This can make it difficult to see the actual changes that were done in the contribution since they are hidden by these formatting changes (P3, P12).

In addition, when receiving comments from a code review, developers can sometimes be sensitive to criticism (P2, P8, P11). This can particularly be the case for new developers: "*If you tell to a new developer that he or she made a mistake, it will be very sensitive. He may feel very uncomfortable because somebody found a mistake in his work. But if a linter tells you about a mistake, it is not a problem, because it's not personal.*" (P8). A new developer might also look up to the person that is conducting the code review which can make the criticism especially dispiriting (P2). Having a linter doing this job can also contribute to people feeling more equal in a project if there is no senior person telling others to do things differently (P11). However, this does not seem to be a common reason for using a linter as only 27% of the survey participants agree or strongly agree with using a linter as a way to avoid giving negative comments to other developers (Figure 5, Avoid giving negative comments).

4.1.6 Learn About JavaScript

ESLint can be used to learn about new JavaScript features or new syntax. P12 used ESLint in helping him to learn the new syntax of ECMAScript 6 (ES6): "*When I switched to ES6, I used it as an educational tool. It's so easy to continue to use var for variable declarations. I used ESLint very strictly to enforce ES6 syntax, otherwise I would probably still use ES5 when I write code. But with the help of the linter it forces you to switch to ES6, which is a good habit.*" (P12).

Even though linters can be beneficial to all JavaScript developers, they can be especially helpful for new developers, either those who are new to a project or those who are new to programming in general (P6, P7, P9, P13, P14). Contributors in OSS projects usually have different levels of experience and using a linter can help with "*leveling the playing field and helping people to understand what's actually going on*" (P13). This particular example came from a developer that had been working with students who were accustomed to getting errors from the Java compiler, telling them what they can and can not do. However when using JavaScript, one can run code that includes various coding mistakes and not get notified about it (P13).

Interestingly, our survey participants did not fully agree with using linters as a way to learn JavaScript: only 31% of them agree or strongly agree, whereas 42% disagree or strongly disagree with it (Figure 5, Learn about JS).

4.1.7 Why Not to Use a Linter?

There were 21 survey participants that had never used a linter in a JavaScript project. These participants only received two additional questions in the survey, including why they do not use a linter. Three participants explained that they simply do not need to use a linter, *e.g.*, as they already use an editor formatter or rather rely on TypeScript for typing analysis and on team discipline and code reviews for code quality. Two participants did not find it sufficiently beneficial to use a linter since it can complicate the build process, can require demanding configuration, and it can increase the cost of the development life cycle. Two others did not have sufficient knowledge about linters, one was not in charge of the repository settings and three simply did not know why they had never used a linter. Other reasons given once were: there is too much effort involved in setting up a linter in a big project, it is difficult to integrate the linter with some IDEs and that other tasks have taken precedence before setting up a linter.

RQ$_1$: JavaScript developers use linters for the following reasons: to maintain code consistency, to prevent errors and augment test suites, to save discussion time about which style to use, to avoid ambiguous and complex code, and to automate code reviews.

4.2 RQ$_2$. Which Linter Configuration Strategies Do Developers Adopt in Their Projects?

In the following we present eight strategies that developers use to configure their linters, which emerged from our qualitative analysis by conducting the interviews. These strategies were presented to our survey participants who could select all methods they had ever used, and also had the chance to add their own methods to the list. The following sections are ordered by their importance, according to the survey participants, which is also displayed in Figure 6.

4.2.1 Use an Existing Preset

There are many publicly available presets that anyone can use in a project instead of creating a custom configuration, or that one can use as a part of a custom configuration. Many of these presets have been carefully constructed by their creators and have been changed attentively over time: "*They thought about the code standard quite extensively and put a lot of thought in it.*" (P12). Several interviewees use a preset as a part of their configuration file (P1, P6, P10, P12, P13, P15) and one participant normally tries to solely use the preset as is (P8).

A large portion of our survey participants (70.2%) also use an existing preset. We also took the opportunity to ask our survey participants about how often they use the default configurations that come with the linter. 51.1% of them affirm to do so. This highlights the importance of such default configurations or presets to be carefully thought, as they are highly adopted by practitioners.

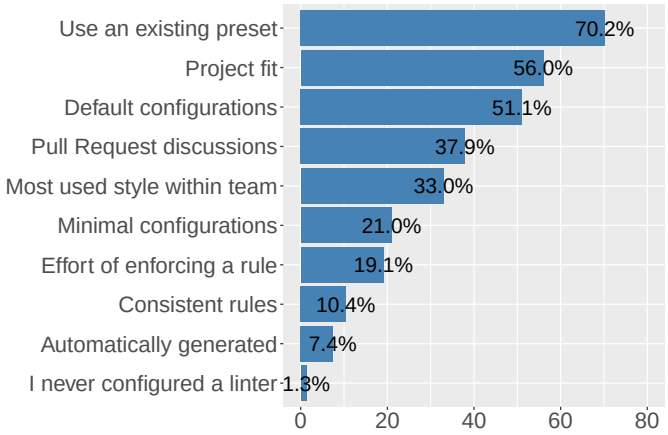


Fig. 6: Strategies used by developers to configure a linter ($N = 309$). The full survey question can be found in our appendix. Number in the bars are represented as percentages.

4.2.2 Project Fit

It is important that the stylistic and best practice rules fit the existing code when the rules are chosen (P3, P4, P6, P12): "*I wanted them to fit the code as it was, I wanted the linting in place with as little mess as possible.*" (P12).

According to P4, if there is already some sense of style in the existing code, it is not very sensible to change it to something else since it would create more work than necessary when setting up a linter. P4 also considers whether a particular rule will be useful for the project or if it will need to be disabled or overridden in multiple locations in the code. If it needs to be disabled frequently, it is not worth it to enforce it.

In our survey, we also observed that choosing rules that fit the current project is a common configuration strategy, as 56% of our participants affirmed to use it.

4.2.3 Pull Request Discussions

Three participants (P2, P4, P10) reported that when something is discussed in a pull request that can be enforced with a rule, they use the opportunity to enable the corresponding rule. According to P2, since the topic surfaced in the pull request, then there was obviously a need to make a decision. That way, the topic will not surface again and time will not be spent on discussing it (P2).

In the survey, 37.9% of the participants affirm to use team discussions as input to choose which rules to enable or disable.

4.2.4 Most Used Style Within Team

P7 reported that in a new project he would most likely go with the code style that is the most common one amongst the developers in the team, consequently adding rules that enforce that style. Generally when working with a new team, the first discussion he often has with the team members is regarding which code style people are used to.

33% of our survey participants also affirm to use the style that is the most commonly preferred one amongst developers of a team.

4.2.5 Minimal Configurations

Some prefer to keep the configuration as simple and minimalistic as possible (P1, P5, P8, P15): "*We don't want people to feel like they have to jump through unnecessary hoops to get their PR's in, so turning on every single thing wouldn't be great.*" (P1). Furthermore, P8 thinks that if too many rules are enabled in a project, people will not trust the configurations: "*They will think that it is a bureaucracy and that it's not important.*" (P8). Both P1 and P8 prefer to only enable rules that can prevent errors.

In the survey, we see that 21% of the survey participants have chosen to also have configurations that are as minimalistic as possible.

4.2.6 Effort of Enforcing a Rule

P15 described that he commonly enables a set of rules, *e.g.*, a known preset, and then sees how it works out for the project. If some of the rules are starting to be bothersome for the project, *e.g.*, needing to be disabled with inline comments or if too much refactoring is required to fit the rule, it is permanently disabled: "*Just start to use it and see how much pain it causes, where it's beneficial. But usually it's turning things off when it's apparent that it's creating more effort than it really helps.*" (P15). He also describes the process as a feedback cycle where it is important that contributors agree on the rules that are used: "*The disagreement between people is very important, you have to get everyone on the same page.*".

Although this was only mentioned by one interviewee, enabling rules that involve less effort to follow is used as a strategy by 19.1% of our survey participants.

4.2.7 Consistent Rules

To some developers it is not important which exact rules are actually enabled (P2, P6, P10): "*I almost don't even care what the rules are, I have some opinions, but I'm much more interested in there just being consistent rules, than having a point of view about any particular rule.*" (P10). There simply has to be some fixed set of rules to enforce consistency and to prevent unnecessary time being spent on discussions: "*Having a linter forces us to make choices, even if it's arbitrary choices in some situations.*" (P2).

Interestingly, only 10.4% of our survey participants affirm that they do not care about which rules are there, as long as they are consistent. We see this as evidence that choosing the right rules matters to most developers.

4.2.8 Automatically Generated

ESLint provides an automatic method to ensure that a configuration fits well to a specific project, based on its source code in addition to a small questionnaire to the developer. Two interviewees used this method to create their configuration file (P11, P14). P11 then reviewed the generated rules and the errors in the output to see if he agreed with them. In general, he does not consider it wise to use a preset since linter configurations are generally very project dependent: "*I didn't even consider Airbnb or Google because I think every project is a little different.*" (P11).

This strategy's lack of popularity among our interviewees is also observed in our survey, where only 7.4% of participants affirmed to use it (making it the least frequent strategy amongst our survey participants).

4.2.9 Additional Methods

Survey participants mentioned 15 additional methods, where three participants claimed to configure the linter according to their personal preferences. Another three participants use a company style guide, where one explained that there is a special committee at his workplace which maintains JavaScript coding standards for the whole company. Other strategies include using the strictest setting possible, choosing rules that prevent bugs or to have a big team discussion on which style to use.

> **RQ_2:** The most common linter configuration strategies are: to use an already existing preset, to choose rules that fit the current style of a project, to enable rules that surface in discussions (*e.g.*, on a pull request), to use the most commonly preferred coding style within a team, to be as minimalistic as possible, and to use rules that involve the least effort to enforce.

4.3 RQ_3. What Linter Rules Do Developers Commonly Enable and Disable?

As stated before, ESLint does not come with default configurations. It therefore has to be configured in some way, most commonly by specifying a configuration file. This configuration can include presets, and/or some rules that are enabled, disabled or set as warnings. We first discuss the prevalence of presets, followed by an analysis on the prevalence of individual rules that are specified.

Throughout the results, we separate the results between projects that make use of a preset as basis and projects that do not make use of a preset as basis. We conjecture their behavior are different: when not using a preset, developers need to think about which rules to enable and which rules not to enable. When using a preset, developers already start with a set of enabled (and, as a consequence, disabled) rules; however, they may disagree with some of the decisions of the preset and, in this case, they have to overrule what was there (in other words, developers make a decision to go "against" the preset). Thus, in our results, we control this possible factor of influence.

4.3.1 Presets

Out of the 9,548 projects that were analyzed, 6,413 or 67.2% used a preset in their configurations, with a total of 6,967 presets being used (some projects using more than one preset). This result is inline with the results of our survey, where 70.2% of participants affirm to use presets as a configuration strategy (Figure 6).

The 10 most popular presets are displayed in Table 7. It is evident that only a handful of presets are extremely popular amongst these projects. Only the five most popular presets account for 67.2% of all presets that are used. The recommended setting from ESLint [31] is the most popular preset, followed by the Airbnb preset [52]. It is interesting that in the top five places, there are two presets by Airbnb: the base version (*airbnb-base* [53]) and the extended version that includes rules for some external JavaScript libraries (*airbnb*).

Preset	Projects	% of all presets
eslint:recommended	2,226	32.0%
airbnb	1,166	16.7%
standard	627	9.0%
airbnb-base[11]	557	8.0%
google	104	1.5%
standard-react	81	1.2%
prettier	58	0.8%
rackt	49	0.7%
react-app	48	0.7%
semistandard	44	0.6%
Total	**4,960**	**71.2%**

TABLE 7: The 10 most popular presets, showing the number of projects using each preset and the percentage of the times the preset is used out of all used presets

If these two presets are counted together, they have a total of 1,723 instances, making *airbnb* and *eslint:recommended* (2,226) by far the most popular ones (56.7% of all presets that are used).

4.3.2 Frequency of Specified Rules

Here we examine the frequency of rules that are specified in configurations, for each type of rule setting and differentiating between projects that use presets and those that do not. The quantity of rules that are configured are displayed in Tables 8a, 8b, 8c, and 8d.

First of all, it is evident that the warning mechanism that is provided by the tool is not often used, as we observe only three warnings used on average per project (with and without presets). We will therefore not focus on this setting in the rest of the analysis. Rules are however most frequently turned on, or on average 17 times per project (with and without presets). In addition, developers also turn off an average of seven rules per project.

It is visible that projects that do not use presets, specify more rules in general than projects that use a preset. More specifically, 95% of the projects that do not use a preset, use at least one rule and specify 58 rules (to be enabled or disabled) on average in their configurations (the remaining 5% only use plugins or do not configure the rules correctly). Furthermore, 70% of the projects that do use a preset have at least one rule specified, and use 10 rules on average. For these projects, the same percentage of projects enable and disable at least one rule (53%) but more rules are enabled on average (seven rules enabled and three rules disabled).

4.3.3 Common Categories

ESLint provides a set of rules that is grouped into seven different categories, as explained in the Background section. We examine which categories are most commonly enabled or disabled, separately between projects that use presets and those that do not. Tables 9 and 10 show the average number of rules that are enabled and disabled, respectively, from each category per project along with how many projects use at least one rule from each category.

11. There were two entries for the Airbnb base configurations, namely *airbnb-base* and *airbnb/base*, with 378 and 179 instances, respectively. The latter is however only a deprecated npm entry point for the previous, so these two were counted as one preset in the analysis.

	Total rules	Mean per proj	# projects	% projects
Projects w/ preset	43,340	7	3,389	53%
Project w/o preset	115,500	37	2,735	87%

(a) Enabled rules

	Total rules	Mean per proj	# projects	% projects
Projects w/ preset	5,159	1	1,125	18%
Project w/o preset	20,122	6	1,601	51%

(b) Warned rules

	Total rules	Mean per proj	# projects	% projects
Projects w/ preset	17,263	3	3,395	53%
Project w/o preset	45,024	14	2,344	75%

(c) Disabled rules

	Total rules	Mean per proj	# projects	% projects
Projects w/ preset	65,762	10	4,465	70%
Project w/o preset	180,646	58	2,979	95%

(d) All rules

TABLE 8: All used rules, divided into the different settings: enabled, disabled and warned rules. Each shows the total number of used rules, the round up arithmetic mean of rules used per project, the number of projects that use at least one rule, and the percentage of projects that use at least one rule. Total number of projects = 9,548, total number of projects with a preset = 6,413, total number of projects without a preset = 3,135.

Enabled Categories. For projects that do not use presets, four categories are enabled in more than 50% of all projects: *Possible Errors*, *Best Practices*, *Variables* and *Stylistic Issues* (Table 9a). The category *Stylistic Issues* is by far the most commonly enabled one, where 82% of all projects that do not use a preset, enable at least one rule. The next category in line is *Best Practices* where 62% of projects without a preset enabled at least one rule. Furthermore, most rules are enabled on average per project for these two categories. However since these two categories also include the highest number of available rules (81 and 69), they are not the most commonly used categories in proportion with the amount of available rules. In that sense, *Possible Errors* is the most commonly used category with 24% of available rules being used on average per project, whereas with *Stylistic Issues* and *Best Practices* the ratio is 13% and 19%, respectively. The other three categories, *Strict Mode*, *Node.js & CommonJS* and *ECMAScript 6*, all have a similar percentage of projects with at least one rule, or around 28-29%.

There is a similar story when it comes to projects that do use a preset (Table 9b). Much fewer rules are generally enabled as shown in the previous analysis on rule frequency (Table 8a), but the most popular categories remain the same.

Category	Avg	% available rules	# proj	% proj
Possible Errors	7.5	24.2%	1,631	52.1%
Best Practices	13.1	19.0%	1,929	61.6%
Strict Mode[12]	0.3	30.0%	895	28.6%
Variables	2.7	22.5%	1,882	60.1%
Node.js & CommonJS	0.9	9.0%	917	29.3%
Stylistic Issues	10.6	13.1%	2,562	81.9%
ECMAScript 6	1.8	5.6%	897	28.7%

(a) Enabled rules per category for projects that **do not use** a preset

Category	Avg	% available rules	# proj	% proj
Possible Errors	0.4	1.3%	777	12.1%
Best Practices	2.1	3.0%	1,171	18.3%
Strict Mode	0.1	10%	448	7.0%
Variables	0.4	3.3%	1,043	16.3%
Node.js & CommonJS	0.2	2.0%	279	4.4%
Stylistic Issues	3.0	3.7%	3,020	47.1%
ECMAScript 6	0.6	1.9%	799	12.5%

(b) Enabled rules per category for projects that **do use** a preset

TABLE 9: Enabled rules per category, showing the average number of rules used per project, the average percentage of rules used per project out of all available rules in the category, the number of projects with at least one rule and the percentage of projects with at least one rule out of all projects. Categories are listed in the order as they appear in the ESLint documentation [28].

Category	Avg	% available rules	# proj	% proj
Possible Errors	1.0	3.2%	1,180	37.7%
Best Practices	3.7	5.4%	1,687	53.9%
Strict Mode	0.3	30.0%	980	31.3%
Variables	0.8	6.7%	1,111	35.5%
Node.js & CommonJS	1.0	10.0%	886	28.3%
Stylistic Issues	6.8	8.4%	2,096	67.0%
ECMAScript 6	0.9	2.8%	532	17.0%

(a) Disabled rules per category for projects that **do not use** a preset

Category	Avg	% available rules	# proj	% proj
Possible Errors	0.3	1.0%	1,569	24.5%
Best Practices	0.6	0.9%	1,505	23.5%
Strict Mode	0.1	10.0%	394	6.1%
Variables	0.2	1.7%	709	11.1%
Node.js & CommonJS	0.1	1.0%	404	6.3%
Stylistic Issues	1.2	1.5%	2,087	32.5%
ECMAScript 6	0.3	0.9%	975	15.2%

(b) Disabled rules per category for projects that **do use** a preset

TABLE 10: Disabled rules per category, showing the average number of rules used per project, the average percentage of rules used per project out of all available rules in the category, the number of projects with at least one rule and the percentage of projects with at least one rule out of all projects. Categories are listed in the order as they appear in the ESLint documentation [28].

Disabled Categories. For projects that use a preset, *Stylistic Issues* is the most commonly disabled category, with 33% of projects disabling at least one rule (Table 10b). The second category in this case is *Possible Errors* with 25% of projects disabling a rule. A singularity in these results is that the *Variables* category is by far the least disabled category out of the four popular ones, with only 11% of projects disabling a rule.

As seen in Table 10a, some projects that do not use presets still disable some rules, even though it does not change any of the linter's functionality. In fact, more than half of all projects that do not use a preset disable at least one rule from the *Stylistic Issues* and *Possible Errors* categories, and disable as much as seven stylistic rules on average. We discuss possible reasons for why rules are disabled in these projects in the next section when examining commonly disabled rules.

4.3.4 Common Rules

Examining which categories are used gives a good overview of which types of rules are commonly used and what developers find important. In this part we dive deeper into these categories and examine which individual rules are enabled and disabled most often.

12. The category Strict Mode is a special case in this analysis since it is the only category that includes only one rule (others have 10 or more rules). For that reason, it is less appropriate to report values for this category on the average percentage of used rules out of all available rules, as that percentage will always be relatively high compared to other categories. To maintain consistency, the values are reported in this table and in the following tables, but are not specifically analyzed in the text.

Enabled Rules. We show in Table 11a that the three most commonly enabled rules for projects without presets, all belong to the *Stylistic Issues* category. These are formatting rules that enforce which type of quotation marks should be used (*quotes*), whether semicolons should be placed at the end of a line (*semi*) and what kind of indentation should be used (*indent*). The next four rules in line are from the *Best Practices* category and *Variables*, namely *eqeqeq* which requires the use of the type-safe triple equality operator (=== instead of ==), *no-undef* which disallows undeclared variables, *no-unused-vars* which disallows unused variables and *curly* that enforces consistent use of curly braces for control statements. More rules in *Best Practices* follow, along with rules from the *Possible Errors* category and *Stylistic Issues*. The rule *no-dupe-keys* prevents errors when two keys in object literals are identical, *no-caller* disallows use of callers and callees which can otherwise make several code optimizations impossible, and *no-unreachable* disallows unreachable code, *e.g.*, after a *return*, *throw* or *break* statement. In general, all rules in this list belong to the four categories that have been popular in the previous analyses: *Possible Errors*, *Best Practices*, *Variables* and *Stylistic Issues*.

Table 11b shows the top 20 enabled rules for projects that do use presets. Interestingly, these rules mostly belong to the *Stylistic Issues* category with 14 out of the 20 rules originating from the category. The same three stylistic rules as before are the most popular ones, followed by *linebreak-style* which enforces consistent line endings, *comma-dangle* which enforces or disallows trailing commas in object literals, and *space-before-function-paren* which enforces consistent use of spaces before function parameter parentheses. These rules are followed by some of the few rules from other categories

	Rule	Category	Freq	% proj
1	quotes	Stylistic Issues	1,898	60.6%
2	semi	Stylistic Issues	1,506	48.1%
3	indent	Stylistic Issues	1,356	43.3%
4	eqeqeq	Best Practices	1,338	42.7%
5	no-undef	Variables	1,270	40.6%
6	no-unused-vars	Variables	1,260	40.3%
7	curly	Best Practices	1,232	39.4%
8	no-dupe-keys	Possible Errors	1,228	39.2%
9	no-caller	Best Practices	1,171	37.4%
10	no-unreachable	Possible Errors	1,163	37.2%
11	no-eval	Best Practices	1,155	36.9%
12	brace-style	Stylistic Issues	1,126	36.0%
13	no-with	Best Practices	1,123	35.9%
14	wrap-iife	Best Practices	1,123	35.9%
15	no-irregular-whitespace	Possible Errors	1,090	34.8%
16	comma-style	Stylistic Issues	1,089	34.8%
17	no-cond-assign	Possible Errors	1,085	34.7%
18	no-func-assign	Possible Errors	1,083	34.6%
19	no-redeclare	Best Practices	1,083	34.6%
20	no-invalid-regexp	Possible Errors	1,079	34.5%

(a) Top 20 enabled rules for projects that **do not use** a preset

	Rule	Category	Freq	% proj
1	semi	Stylistic Issues	1,641	25.6%
2	indent	Stylistic Issues	1,567	24.4%
3	quotes	Stylistic Issues	1,336	20.8%
4	linebreak-style	Stylistic Issues	888	13.8%
5	comma-dangle	Stylistic Issues	714	11.1%
6	space-before-function-paren	Stylistic Issues	605	9.4%
7	eqeqeq	Best Practices	601	9.4%
8	no-unused-vars	Variables	595	9.3%
9	curly	Best Practices	515	8.0%
10	no-trailing-spaces	Stylistic Issues	491	7.7%
11	brace-style	Stylistic Issues	482	7.5%
12	strict	Strict Mode	451	7.0%
13	space-before-blocks	Stylistic Issues	424	6.6%
14	max-len	Stylistic Issues	419	6.5%
15	no-use-before-define	Variables	419	6.5%
16	keyword-spacing	Stylistic Issues	416	6.5%
17	object-curly-spacing	Stylistic Issues	396	6.2%
18	eol-last	Stylistic Issues	386	6.0%
19	new-cap	Stylistic Issues	366	5.7%
20	no-eval	Best Practices	365	5.7%

(b) Top 20 enabled rules for projects that **do use** a preset

TABLE 11: Top 20 enabled rules for projects that use and do not use a preset, showing the total number of projects using them, along with the percentage of those projects out of all projects that use or do not use a preset

which were also detailed in the previous list.

Disabled Rules. Table 12b shows the most commonly disabled rules for projects that use a preset. The rule *no-console* from the *Possible Errors* category is by far the most commonly disabled rule, with 19% of projects specifying the rule. Logging with the console is mainly used for debugging purposes which should normally not be present in production code. This is followed by *no-underscore-dangle* from the *Stylistic Issues* category which disallows dangling underscores in identifiers, which are commonly used to indicate a private variable. Interestingly, here we observe again *comma-dangle* as the third one on the list, thus proving to be a controversial rule. Other rules include *func-names* to require or disallow named function expressions for debugging purposes, and *no-param-reassign* from the *Best Practices* category which disallows reassigning function parameters.

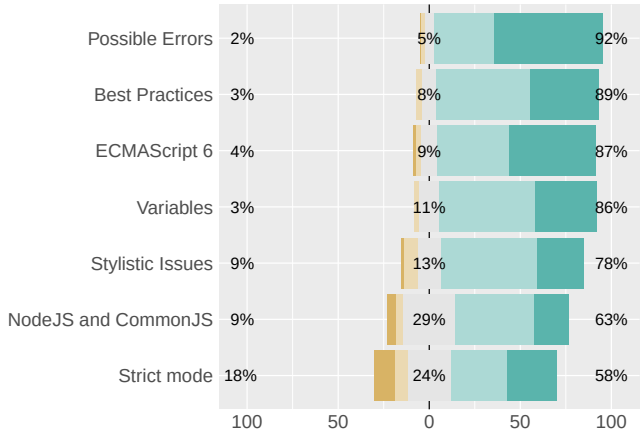


Fig. 7: Perception of participants on the importance of each category. Bars show Unimportant, Slightly important, Neutral/Not Applicable, Important, and Very Important, respectively. ($N = 266$)

Interestingly, as before, we also observed some rules being disabled even in projects that do not use a preset (Table 12a). We conjecture two possibilities to why that happens: 1) Some configurations are written in a way where every single available rule is listed, and each is set to being enabled or disabled. This might make changes clearer, for example showing which rules existed when the configuration file was last updated, and thus differentiating between newly added rules by ESLint and older rules that the developers decided to later enable, and 2) Developers want to explicitly show that some rules should absolutely not be turned on. For example, the rule *no-underscore-dangle* is disabled by 49.3% of projects that do not use a preset, where developers might want to show that they indeed allow the naming pattern where variables start with an underscore. This would make the style obvious to a newcomer and it would also be clear that the rule was not simply forgotten or overseen by not being included in the configurations. Future research needs to be conducted in order to better understand why this behavior occurs.

4.3.5 Developers' Perceptions on the Importance of Categories and Rules

The survey participants were asked which ESLint categories they considered important to include in configurations. Figure 7 shows that *Possible Errors* is the most important category where 92.5% of the participants consider it to be either important or very important. *Possible Errors* is then followed by *Best Practices*, *ECMAScript 6*, and *Variables*, where over 86% of the participants consider each of them to be either important or very important. For these four categories, only four participants considered any of them to be unimportant (one for *Possible Errors* and three for *ECMAScript 6*). The three remaining categories, *Strict Mode*, *Node.js & CommonJS* and *Stylistic Issues*, are all widely considered to be important as well, but to a lesser extent than the other categories. *Strict Mode* is the category that is most commonly considered to be unimportant by the participants (11%).

Interestingly, we observe that there is a mismatch between the perceptions of participants on how important

	Rule	Category	Freq	% proj
1	no-underscore-dangle	Stylistic Issues	1,543	49.3%
2	strict	Strict Mode	982	31.4%
3	no-ternary	Stylistic Issues	686	21.9%
4	func-names	Stylistic Issues	681	21.8%
5	new-cap	Stylistic Issues	634	20.3%
6	one-var	Stylistic Issues	608	19.4%
7	sort-vars	Stylistic Issues	597	19.1%
8	padded-blocks	Stylistic Issues	582	18.6%
9	no-use-before-define	Variables	581	18.6%
10	consistent-return	Best Practices	572	18.3%
11	no-console	Possible Errors	570	18.2%
12	camelcase	Stylistic Issues	565	18.1%
13	no-extra-parens	Possible Errors	560	17.9%
14	func-style	Stylistic Issues	549	17.5%
15	no-plusplus	Stylistic Issues	535	17.1%
16	vars-on-top	Best Practices	525	16.8%
17	no-shadow	Variables	525	16.8%
18	no-warning-comments	Best Practices	514	16.4%
19	valid-jsdoc	Possible Errors	499	15.9%
20	wrap-regex	Stylistic Issues	495	15.8%

(a) Top 20 disabled rules for projects that **do not use** a preset

	Rule	Category	Freq	% proj
1	no-console	Possible Errors	1,216	19.0%
2	no-underscore-dangle	Stylistic Issues	646	10.1%
3	comma-dangle	Stylistic Issues	479	7.5%
4	func-names	Stylistic Issues	430	6.7%
5	no-param-reassign	Best Practices	411	6.4%
6	strict	Strict Mode	395	6.2%
7	no-use-before-define	Variables	358	5.6%
8	consistent-return	Best Practices	356	5.6%
9	max-len	Stylistic Issues	315	4.9%
10	padded-blocks	Stylistic Issues	305	4.8%
11	arrow-parens	ES6	266	4.1%
12	new-cap	Stylistic Issues	262	4.1%
13	camelcase	Stylistic Issues	259	4.0%
14	global-require	Node.js	254	4.0%
15	no-shadow	Variables	243	3.8%
16	arrow-body-style	ES6	224	3.5%
17	no-plusplus	Stylistic Issues	204	3.2%
18	vars-on-top	Best Practices	196	3.1%
19	space-before-function-paren	Stylistic Issues	194	3.0%
20	id-length	Stylistic Issues	193	3.0%

(b) Top 20 disabled rules for projects that **do use** a preset

TABLE 12: Top 20 disabled rules for projects that use and do not use a preset, showing the total number of projects using them, along with the percentage of those projects out of all projects that use or do not use a preset

each category is and the number of projects that actually enable these categories (Figure 7 and Table 9). More specifically, the *Possible Errors* category is considered important by 92.50% of our participants, while only 52.10% of the projects actually enable at least one rule in the category (among projects that do not use a preset). Similar ratios are found for *Best Practices* and *Variables*, and an even larger difference exists for *ECMAScript 6*. We also highlight an interesting fact about the *Stylistic Issues* category: while this category is by far the most enabled one in projects (81.9% of projects that do not use a preset and 47.1% of projects that do use a preset enable at least one of its rule), it is only the fifth most important category according to the developers' perceptions (although considered important or very important by 78% of participants).

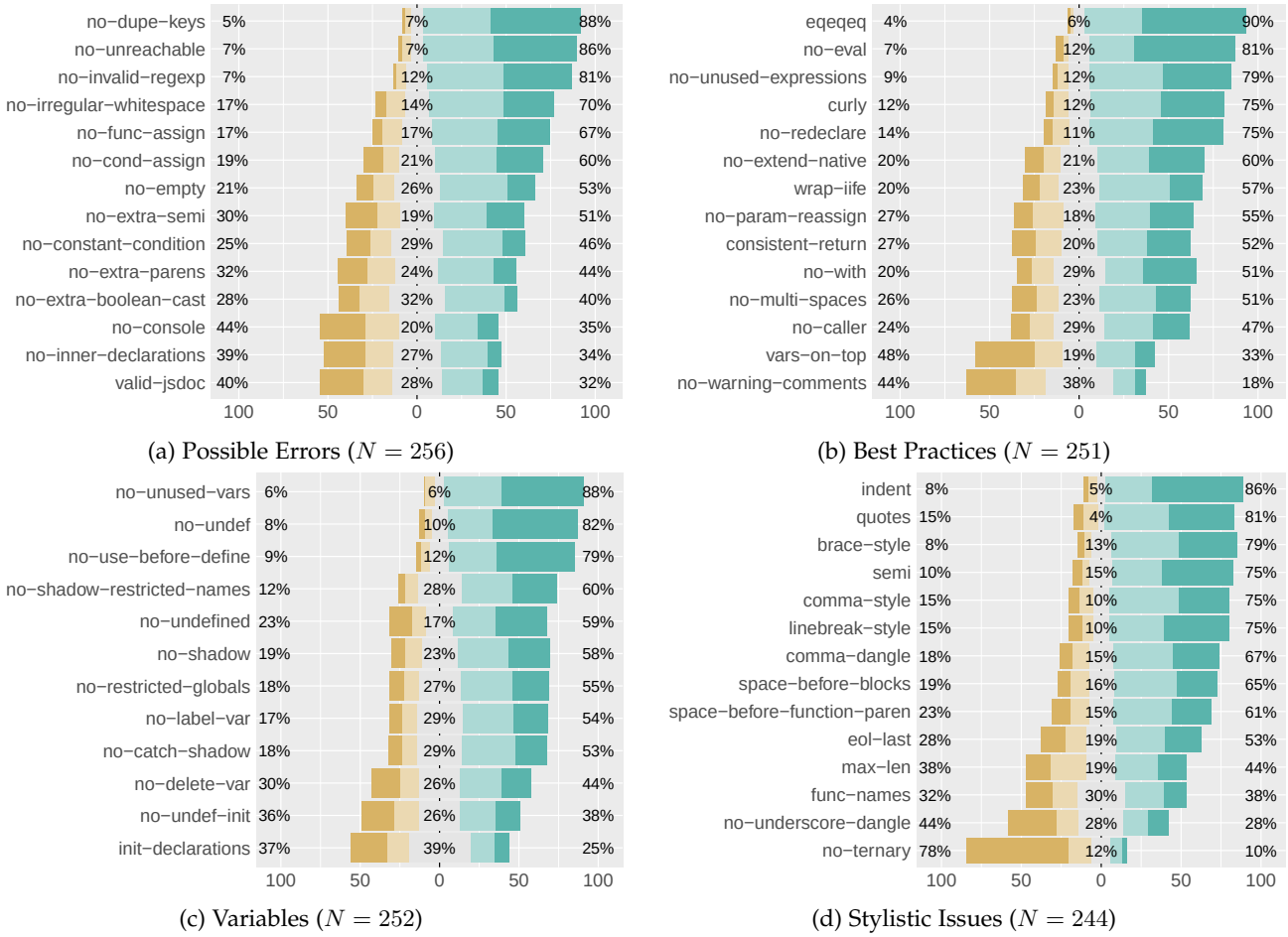
We hypothesize a few reasons for such a phenomenon, which should be further studied: 1) Although participants perceive these categories as highly important for a linter, they do not completely provide what developers need, or 2) Participants are not actually spending time configuring their linters to provide feedback on what they believe is important.

To know which individual rules are considered to be the most important ones, we asked the survey participants to rate the importance of a chosen set of 54 rules. By mining projects on GitHub we saw that there are four categories that are most commonly used in configurations: *Possible Errors*, *Best Practices*, *Variables* and *Stylistic Issues*, and these were also discussed most often in the interviews. For each of these categories we asked the participants to rate the importance of 14 rules, where the individual rules were chosen as described in Section 3.3. We show the results in Figure 8.

The rules in the *Possible Errors* category are shown in Figure 8a. Using the median rating value, eight of the 14 rules are thought to be either important or very important. The rules that originate from the list of enabled rules without using a preset are generally thought to be the more important ones, where *no-dupe-keys* is the most important rule, followed by *no-unreachable* and *no-invalid-regexp*. Moreover, the rule *no-console* seems to be one of the most debatable rules where it is thought to be unimportant by 25.6% of participants but also important by 35.2% of participants. This rule was further commented on by four survey respondents, saying that it is sometimes not necessary to include the rule on single-person projects and that most logging should be disallowed except for logging errors. In addition, one participant mentioned the importance of the rule *no-await-loop* (not included in list) due to potential performance problems. Others claimed that the rules that cause actual bugs are generally the most important ones (without mentioning any specific rules).

The *Best Practices* rule set is shown in Figure 8b. In this case, most rules are thought to be (very) important or 12 out of the 14 listed rules. The rules *eqeqeq* and *no-eval* appear to be the most important ones where 90.4% and 81.1% consider them to be either important or very important. These are then followed by *no-unused-expressions* and *curly*. Furthermore, the rules *vars-on-top* and *no-warning-comments* are notably the least important ones, which were originally retrieved from the lists of the most commonly disabled rules in GitHub projects. One participant shared his discontent with the rule *no-warning-comments*, saying that he did not know about it previously and that he "*would probably flip a table if it broke a CI build*". Another participant expressed that this rule should only be a part of a specific setup for deployment or production and not for development.

A general observation was also shared by another participant saying that it is good to enforce some rules for production code but that they "*should not hinder one's development style*". The less rated rule *vars-on-top* was also said to be unimportant as with ES6 it is possible to have block scope local variables with *let* and *const* (instead of using the *var* statement which defines a variable regardless of block scope). Lastly, one respondent discussed the rule *no-extend-native* and the two opposing views about extending native objects. Being in favor of the feature, he described it as one



(a) Possible Errors ($N = 256$)

(b) Best Practices ($N = 251$)

(c) Variables ($N = 252$)

(d) Stylistic Issues ($N = 244$)

Fig. 8: Participants' perceptions on the importance of different rules. Bars show Unimportant, Slightly Important, Neutral/Not Applicable, Important, and Very Important, respectively.

of the advantages of JavaScript where avoiding it is "*akin to leaving the protective plastic on a couch*".

For the *Variables* category in Figure 8c, the set simply consists of all available rules as there are only 12 in the category. Nine out of the 12 rules are considered to be important by the participants. Three rules seem to be particularly important, *no-unused-vars*, *no-undef* and *no-use-before-define*, where 87.5%, 81.9% and 79.4% of the participants, respectively, consider them to be either important or very important. No specific comments were made about any of these rules.

The subset of rules for the *Stylistic Issue*s category is shown in Figure 8d. 10 out of the 14 rules are considered to be important, but those are exactly the rules that were retrieved from the lists of the most commonly enabled rules. One rule is considered to be very important (based on the median value), *indent*, where 86.3% consider it to be either important or very important. There are three other rules which over 40% of the participants consider to be very important: *quotes*, *semi* and *linebreak-style*. On the contrary, the rule *no-ternary* is unquestionably the least important rule where 63.8% of participants consider it to be unimportant and only 9.6% consider it to be important. Interestingly, this rule had only been enabled 8 times in the 3,135 (0.3%) analyzed projects that do not use a preset, despite having been available for 2,5 years at the time of analysis. Lastly, the rule *comma-dangle* was mentioned by two participants to be particularly good to catch errors, where it was described as "*unquestionably the source of most of my lint errors*"[13]. Other participants generally described rules from this category to be beneficial for code readability within a team and for simple code merging.

RQ$_3$: Presets are used by 67% of all ESLint projects, and projects that do not use presets specify a high number of 58 rules on average, whereas those that do use presets specify 10 rules on average. Stylistic Issues is the most commonly enabled category, followed by Best Practices and Variables. The Possible Errors category is considered to be the most important one, according to survey participants.

13. The rule *comma-dangle* was previously a part of the *Possible Errors* category due to the known errors that dangling commas could cause in Internet Explorer version 8 and below. The rule was however moved to the Stylistic Issues category when older versions of Internet Explorer stopped being supported by Microsoft, thus making this error less relevant.

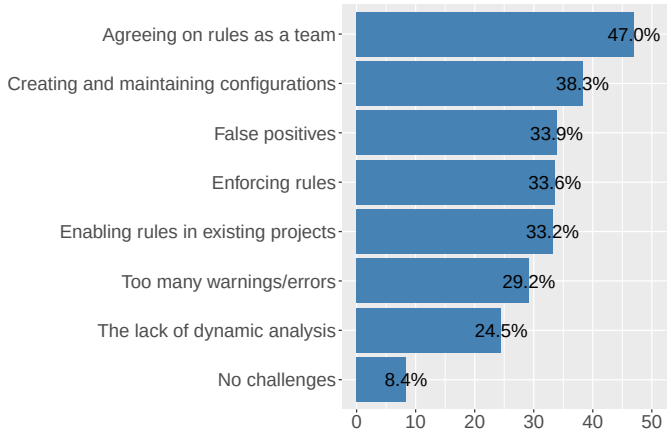


Fig. 9: The challenges that participants face when using a linter ($N = 298$). The full survey question can be found in our appendix. Numbers in the bars are represented as percentages.

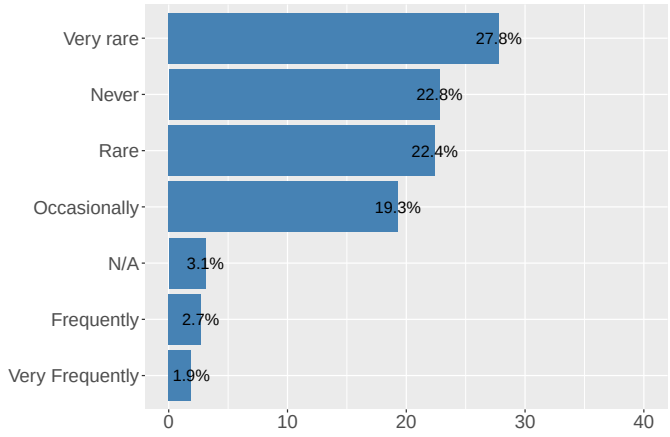


Fig. 10: The frequency of how often developers experienced false positives when using ESLint.

4.4 RQ$_4$. What Are the Challenges in Using a JavaScript Linter?

In this sub-section, we discuss the results that we derived from the interviews, together with the agreement of our survey participants. We show the most commonly faced challenges in Figure 9. We provide one additional option in the survey that did not emerge from the interviews, namely: *too many warnings/errors outputted from the linter*, as it was suggested by one of the pilot test participants and has been reported in related research to be a common problem with using static analysis tools [8], [54].

In the following, sections are ordered by their frequency, according to the survey participants.

4.4.1 Agreeing on Rules as a Team

In an open source project it is relatively easy to build consensus around what people want to do (P4, P15). If someone wants a new rule to be enabled, then only a few main contributors need to say yes and it is decided upon. Other maintainers usually follow what the project leaders propose (P4, P15). In a business environment, where there may not be clear leaders, several members of the team can have different opinions on which rules to enable and disable: "*There are 60 people with their own opinion about how code should be written.*" (P4). P4 further explains that it can be especially difficult to introduce rules on stylistic issues since it is not considered to be important and it can be hard to justify why you would inconvenience people to adhere to new rules: "*It may create more tension than it does actually solve problems.*" (P4).

Agreeing on which rules to use within a team is also seen as a challenge for our survey participants, but almost half of them (47.0%) reported to have experienced it as a problem.

4.4.2 Creating and Maintaining Configurations

Several interviewees mentioned that it was challenging to create the original configuration or to keep it updated (P1, P3, P5, P8, P12). Only two participants reported that they had read over all available rules when they originally created the configuration file (P3, P9). This involves evaluating a set of 236 rules that ESLint has available which can be a tedious process: "*Most of it was read through every rule, it was kind of a very painful process to set it up.*" (P3). Another participant used very similar wording when it came to setting up the tool: "*Sometimes a pain to set up in your editor with the right configuration.*" (P5). Meanwhile, the other participant that manually created the configuration file claimed that it was actually quite easy to set up (P9).

Others have been frustrated with keeping their configurations up to date after they have been created, especially when using presets (P1, P8, P12). When the presets are updated, there are often new rules that are enabled or older rules that are changed which can cause a high volume of new warnings or errors (P12). P12 explained that these changes are sometimes beneficial and he happily fixes the warnings, but at other times the change is not useful and they have to be overwritten. Moreover, P1 discussed that it can be frustrating when the presets update very frequently and the code has to be changed often because of it. However he also likes that the presets keep him updated of new rules regarding new JavaScript features.

A substantial portion of our survey participants also face these challenges: 38.3% agreed that creating or maintaining configurations was a challenging part of using a linter.

4.4.3 False Positives

We also questioned our survey participants about how often they experience false positives (as, according to our interviewees, this does not happen often, contradicting related work [55], [8], [10], [56]).

Interestingly, although one third of the participants see false positives as a challenge (see Figure 9), their perception on how often they observe false positives when using ESLint is low, as we see in Figure 10.[14] We discuss more about this contradiction in Section 5.3.2.

4.4.4 Enforcing Rules

As previously introduced in the Background section, there are three possible settings for a rule in ESLint: *off*, *warn*

14. We also asked how often users from other JavaScript linters encounter false positives, and numbers were also similarly low. As we focus on ESLint in this paper, these other numbers can be found in our online appendix [35].

and *error*. The majority of the interviewees used only or mostly errors and rarely warnings (which also matches with our findings in GitHub repositories, see Table 8). These settings are used by the participants for different purposes, such as indicating the criticality of a rule (P15) or by using warnings as an adaptation period when enabling new rules (P4, P7, P13). Other participants liked to use warnings in the development process so that their build would not be interrupted when working on unfinished features (P9, P12, P15).

There is however a problem with using warnings: several interviewees claimed to use only errors since warnings would simply be ignored by developers (P2, P3, P4, P13). In addition, according to these participants, when warnings live for a long time in the codebase, people start to devalue them and leave them behind. Developers do not feel any responsibility for the warnings and might think "*there were some warnings when I checked it out, not my job to fix it, I'm going to check it back in with warnings.*" (P4). P4 further explains that especially if there are many warnings, developers will not even read them and simply leave them behind. To enforce removal of the warnings, they therefore have to be set as errors that actually prevent a build from succeeding.

Indeed, making sure that the defined rules are actually followed by developers is a challenge that many developers face. In our survey, 33.6% of the participants had trouble with enforcing developers to follow the defined rules in the past.

4.4.5 Enabling Rules in Existing Projects

Six participants (P1, P3, P4, P7, P10, P13) mentioned that it can be difficult to start using a linter or to enable new rules in an existing codebase. Moreover, 33.2% of our survey participants also face this challenge.

If the rules that are enabled cause many warnings or errors to occur, substantial effort is needed to go over all existing code to fix every reported instance. There can even be such a large volume of existing code that it would simply take too much time to review: "*The problem with [the project] is that it's really old and big, so for that we don't have the luxury of turning on whatever rules we want to because we're never going to be able to update a lot of our code to support them.*" (P1).

Even if it is possible to fix all warnings, it can be risky to change old code to conform to these rules since it is easy to introduce new bugs in the meantime (P3, P7, P10). Knowing the original intent of the code can be very difficult and can take considerable time to try to understand (P7). P7 discussed an example where there were many instances of two equal marks being used in the code instead of three, which resulted in many errors from the linter. In these cases it could have been intentionally written so or by accident, which would need to be carefully verified and tested. The risk and effort of going back and changing the old code might therefore not be worth it: "*Cleaning up for just clean up, just to enable it, I think is risky. I've been bitten by that a couple of times.*" (P3). Enabling a linter in a project to begin with can be a lot easier: "*A linter is great if you start with that and you enforce all those rules and the idea is that you will never run into ambiguous code.*" (P7). However with older projects it can be more difficult and even dangerous (P7).

In large existing projects with many collaborators it can also cause conflicts and frustration when new rules are enabled (P4, P13). P4 was working at a company with around 60-70 developers, all working on the same code every day. Usually there are multiple pull requests open at the same time and there will be many merge conflicts when a new rule is suddenly enabled, since it will likely affect code everywhere in the project. In P13's project it caused frustration with people to enable many rules all at once since the developers were not accustomed to them before. Instead, it was decided to do it slowly by enabling only one rule at a time to give developers a chance to get used to the new rules.

4.4.6 The Lack of Dynamic Analysis

JavaScript has been described as harsh terrain for static analysis because of its dynamic nature [57]. Static analysis for JavaScript has thus been critized and said to be limited since it can not account for runtime behavior [58]. The majority of the interview participants would indeed like ESLint to be able to do more, but they however think that the current version is very acceptable since they choose themselves to work with a dynamic language (P2, P5, P6, P7, P8, P9, P10, P11, P15). When P9 was asked whether he misses dynamic analysis from the linter he replied: "*Of course, but I don't particularly think that is ESLint's fault, so much as a language defect. Due to the dynamic nature of JavaScript, static analysis ranges from extremely difficult to impossible. I don't expect ESLint to be able to fix that.*" (P9).

Regarding type checking, there are tradeoffs in choosing JavaScript as a programming language and there is a reason why some languages are not strictly typed (P2, P15). If static typing is something that a developer wants, he or she should rather use TypeScript or some statically typed language (P2, P10, P15): "*Each developer has to make a decision on if they want to work in a typed language or in an untyped language and the tradeoffs of that kind of lead you to the path of what tools can provide.*" (P15). The majority therefore claimed to be quite satisfied with what the linter can achieve and do not expect it to be able to detect more of the dynamic side of the language.

Other participants were more bothered by the fact that a linter can not analyze the dynamic parts of the language, where the problem was mostly centered around the lack of variable types (P3, P4, P13, P14). P3 reports that he has spent substantial time on testing various mix-ups with strings and numbers, for which he would either like ESLint to warn about types that change or would like to switch from using regular JavaScript to TypeScript. Indeed, we observe that, in order to remedy this, organizations like Microsoft and Facebook have exerted substantial effort on, *e.g.*, TypeScript [59] and Flow [60].

24.5% of our survey participants see the lack of analysis for the dynamic features of the JavaScript language as a challenge. These numbers strenghten the motivation for more research on the topic.

4.4.7 Additional Challenges

Participants also had the opportunity to add other challenges which 15 of them did. Amongst these were three challenges mentioned by more than one participant: 1) Getting

team members to use a linter (3 participants), 2) Integration with an IDE or other tools (3 participants), and 3) When ESLint, presets or plugins change (2 participants). This last challenge was described by one participant as: "*I detest it if the rules in a preset change*", which is largely captured by a previous challenge in Section 4.4.2. Other reported challenges included difficulty with writing new linting rules and the fact that using the tool during development is "*obnoxious*" but is more beneficial when it comes to the time of committing code.

> **RQ$_4$:** The challenges that developers face when using linters are: to agree on rules in an industrial setting, to create and maintain the configurations, the (low but existing) number of false positives, to enforce developers to follow the rules, to enable rules in existing projects, and the lack of analysis for JavaScript's dynamic features.

5 Discussion

The results have several implications for developers, linter creators and researchers, which we discuss in the following. Finally we explore the relation between the results and the experience of the survey participants.

5.1 Supporting Developers in Configuring Their Linters

There are mainly two ways of defining which rules can be configured by developers: 1) by using an existing preset, or 2) by manually selecting rules.

Although presets have been developed by experts from the JavaScript community, our study suggests that making use of a preset in an existing project can be challenging to developers as the number of warnings can be high. Thus, in cases where greenfield engineering is not possible, developers should enable rules carefully and incrementally. By incrementally introducing rules and continuously fixing the corresponding warnings, the number of warnings can be kept to a minimum, making developers more likely to examine and fix them. Previous research has also reported that many static analysis tools output too many warnings which makes it more difficult to use these tools [8], [61], [4].

We recommend developers to start by enabling the rules that they consider as the most important ones. After analyzing the three different sources of data, we conjecture that some rules are indeed more important than others (*i.e.*, are used and considered important by several projects and developers). More specifically, we collected the top four[15] rules that appeared in the three data sources: they were mentioned by the interviewees, appeared on the top 10 most frequent rules in GitHub, and received a median rating of "Important" in the survey. As we did not discuss with or ask any of the participants about the rules in the categories ECMAScript 6, Node.js & CommonJS, and Strict Mode, we did not make any assumptions or recommendations regarding those.

15. For the *Variables* category, we were able to collect only three rules.

These rules are shown in Table 13, in no particular order except being grouped by category. When compared to the ESLint recommended, Airbnb, Google, and Standard presets (the four most popular presets in our sample, see Table 7), we observe that the rules we propose are only fully covered by the Standard preset. We suggest developers to take this list as a starting point for teams that face the challenge of agreeing on which rules to use, where these rules have been shown to be important to many other developers. While this is the first step towards an empirical catalogue, research needs to be conducted so that this catalogue can go beyond developers' perceptions (as we also discuss in Section 5.3.4).

Additionally, we recommend developers to use presets from the very beginning in case of greenfield engineering, where developers still do not have to deal with large complex codebases or legacy issues. As aforementioned, the existing presets contain the set of rules that experts from the JavaScript community believe to be the most important ones. Thus, presets can serve as an initial point for new projects, where the number of warnings is still manageable even with the possibly large number of enabled rules in them. In particular, Ayewah *et al.* [7] found that FindBugs users are more likely to fix warnings that apply to new code than to older code. We expect the same phenomenon to happen in greenfield JavaScript systems.

5.2 Implications for Tool and Preset Makers

We have seen the main reasons as to why developers choose to use linters along with which categories they find to be the most important ones. Tool creators should therefore place emphasis on these topics that developers are most concerned with. Both the interviewees' and survey respondents' main reasons for using linters were to maintain code consistency and to catch possible bugs in code. While stylistic rules are used the most often in configurations, developers claim that bug-catching rules are far more important. These two topics should therefore be of main interest for tool makers when deciding on what type of functionality to implement.

Table 13 displays the rules that are deemed the most important based on this study's results. This list of rules could be further expanded and ordered based on the rules' importance, which could be used by ESLint to assign a priority to all available rules. It has been shown in this study that it can be difficult to configure a linter, especially when one needs to go through all available rules to choose from. Partly for that reason, the majority of developers choose to use a preset in their configurations, while also making some modifications to their settings. The process of going through all 236 ESLint rules could be made easier by assigning them with an importance rating which could be based on this study's results, *i.e.*, on the importance reported by the JavaScript community and the frequency of the rules being enabled in GitHub projects.

On a similar note, we can also identify some rules that are rarely used or commonly disabled and are not thought to be important by the JavaScript community. These rules include *no-ternary* (*Stylistic Issues*), *no-underscore-dangle* (*Stylistic Issues*) and *vars-on-top* (*Best Practices*). Preset creators should make sure to not enable these rules as most developers do not want them to be used.

Rule	Category	ESLint recommended	Airbnb	Google	Standard
no-dupe-keys	Possible Errors	✓	✓		✓
no-unreachable	Possible Errors	✓	✓		✓
no-invalid-regexp	Possible Errors	✓	✓		✓
no-irregular-whitespace	Possible Errors	✓	✓	✓	✓
no-unused-vars	Variables	✓	✓	✓	✓
no-undef	Variables	✓	✓		✓
no-shadow-restricted-names	Variables				✓
eqeqeq	Best Practices		✓		✓
no-eval	Best Practices		✓		✓
curly	Best Practices			✓	✓
no-caller	Best Practices			✓	✓
semi	Stylistic Issues		✓	✓	✓
indent	Stylistic Issues		✓	✓	✓
quotes	Stylistic Issues		✓	✓	✓
comma-dangle	Stylistic Issues		✓	✓	✓

TABLE 13: Most important rules to include in configurations and their existence in the ESLint recommended [31], Airbnb [62], Google [63], and Standard [64] presets.

In general, preset makers should be careful to not update the rule selection too often. Developers that use the presets can get frustrated when they have to do frequent changes to their code due to the presets being modified.

Furthermore, as it can be difficult to enforce linting rules without making them break the build, other methods would be beneficial to encourage developers to fix warnings. Sadowski *et al.* [65] had a similar experience when introducing static analysis tools at Google, where developers would ignore warnings that did not cause the build to fail. Tool creators should therefore find and employ other ways to make developers feel responsible to fix them. For example, this could be done by linking the output of the tool to the project repository, to make a somewhat more personal connection to the developers that are responsible. A warning could perhaps be marked with the name of the developer who introduced it, which could motivate him or her to fix the warning.

5.3 Implications for Researchers

The results offer ample opportunities for further research into these findings. Example research directions are listed below.

5.3.1 Evolution of Linter Configurations Over Time

Beller *et al.* [3] studied in large scale the configurations of static analysis tools and how those configurations change over time. Since several participants claim that it is difficult to enable linters for existing code, it would be intriguing to further research how configuration files in projects that enable a linter early on differ from those in projects that enable a linter at a later stage. Furthermore, many participants explained that they try to create the configuration so that it fits the project in the best way possible. It would be valuable to know how thoroughly projects follow these configurations, providing insight into how well the configuration reflects the project style and how much developers care about upholding these code standards.

5.3.2 False Positives

One third of the survey participants affirm that false positives are a challenge when using linters. On one hand, these findings are in line with what previous literature reports on general static analysis tool usage [55], [8], [10], [56], that the presence of false positives is indeed a problem. On the other hand, both our interviewees and survey respondents claim to not often experience false positives when using ESLint (see Figure 10). These findings are opposite to what previous literature reports on general static analysis tool usage [55], [8], [10], [56], where false positives are said to be frequent. We conjecture this might be due to the relatively simplistic analysis methods that are applied in linters and the non-complex issues they mostly identify.[16]

Our current data does not show which rules provide developers with false positives. However, our intuition suggests that researchers should separate rules that apply simplistic analysis (such as identifying the usage of tabs or spaces) from rules that apply more sophisticated analysis (such as identifying non used variables). We do not expect the former to be the main cause of false positives. Thus, more research should be conducted in order to better understand the quantity of false positives in JavaScript linters.

Additionally, research shows that the fact that developers tend to categorize irrelevant warnings as false positives [11], [65] might contribute to the number of experienced false positives. ESLint, in particular, offers vast functionality for configuring the tool to fit developers' needs and preferences, and we've seen that ESLint users often take advantage of that. Research needs to be conducted in order to evaluate whether ESLint users experience fewer false positives because they have configured the tool appropriately.

5.3.3 Reasons for Not Using a Linter

Similarly to Johnson *et al.* [8], we have studied the challenges in using a linter by interviewing active users of such a tool. Furthermore, much like Christakis and Bird [6], we surveyed linter users and asked which challenges they face. In our survey, most participants had used a linter but we do not know how actively they used one or if they continued using a linter. As the main focus of our survey was to explore the experiences of linter users, we did not ask many

16. In our ASE paper [20], we affirmed that these results *only* contradicted previous literature. In this paper, in the light of new data, we observe that false positives are still an open question in JavaScript linters.

questions to the few participants that had never used one. It would be informative to specifically target developers that do not use a linter or those that previously used a linter but have ceased using one. These developers might provide different insights into the challenging parts of using such a tool.

5.3.4 Evidence-based Linting Configuration

Our results shed light on what rules developers see value in and what rules they do not, by means of two different points of view: first, the configuration they actually choose to apply to their projects (more than 9,000 projects), and second, based on their perceptions (collected after a survey with more than 300 developers).

Currently, all the most popular presets are based either on the communities' or companies' own experiences. Our proposal as to which rules to enable (in Table 13) is the first set of rules for ESLint that comes from concrete data.

Although our results do not provide an explanation of what makes developers decide between one rule or the other (as previously discussed), our results emerge from the experience (and choices) of thousands of projects. In future research, we plan to evaluate in which scenarios these choices are the best ones; however, we again reiterate that our results provide a very first concrete step towards evidence-based linting configuration.

5.4 The Role of Experience

Throughout the analysis of our survey, we put all participants in the same bucket, regardless of their experience as software developers. However, experience can indeed be a factor of influence, as studies have shown that experience, among other examples, can play a role in productivity [66] and in testing [67].

To investigate the role of experience in our results, we applied statistical methods that would reveal any differences between groups. To separate participants in these different groups, we first measured the first, second, and third quantiles of their years of experience in software development ($1Q = 4$, $Median = 4$, $3Q = 10$). With that, we derived four groups, where each group contained participants within the following range of experience: 1) G1 = $[0, 1Q]$ (116 participants), 2) G2 = $]1Q, Median]$ (75 participants), 3) G3 = $]Median, Q3]$ (83 participants), and 4) G4 = $]Q3, \infty[$ (84 participants).

We then set a number of *hypotheses*:

- H_1. The experience of participants does influence their motivations as to why they use linters (related to RQ_1).
- H_2. The experience of participants does influence their configuration strategies (related to RQ_2).
- H_3. The experience of participants does influence their perceptions on which ESLint rule categories are most important (related to RQ_3).
- H_4. The experience of participants does influence the challenges they face (related to RQ_4).

We applied a Chi-Squared test [68] to test each of the hypotheses. The test is often used to determine whether there is a significant difference between the expected frequencies and the observed frequencies in one or more categories. We used a significance level of $\alpha = 0.05$ and all the scripts can be found in our online appendix [35]. Although we perform multiple tests, we did not apply any correction on the significance level (*e.g.*, Bonferroni correction). The reason is that we are interested in the output of each individual test, and not that that all hypotheses are true simultaneously [69].

For H_1, we applied the Chi-Squared test for the combination of the four different experience groups and five options in the Likert scale (strongly disagree, disagree, neutral/NA, agree, strongly agree). We also repeated the same procedure for each one of the seven reasons (see Figure 5). We did not obtain a significant *p-value* in any of the tests, which means we are not able to accept the hypothesis that the experience influences any of their motivations.

For H_2, we applied the Chi-Squared test for the combination of the four different experience groups and whether that participant has applied that configuration strategy (yes/no). We also repeated the same procedure for each one of the nine strategies (see Figure 6). We obtained a significant p-value for two strategies: to fit the configuration to the style of the project, and to decide on rules after discussing them in pull requests or similar mechanisms. In both, it seems that, the more experienced a developer is, the more he or she makes use of these strategies.

For H_3, we applied the Chi-Squared test for the combination of the four different experience groups and five options in the Likert scale (unimportant, slightly unimportant, neutral/NA, important, very important). We also repeated the same procedure for each one of the seven rule categories (see Figure 7). We did not obtain a significant *p-value* in any of the tests, which means we are not able to accept the hypothesis that the experience influences their perceptions on which rule categories are most important.

Finally, for H_4, we applied the Chi-Squared test for the combination of the four different experience groups and whether that participant has faced that challenge (yes/no). We also repeated the same procedure for each one of the seven challenges (see Figure 9). We did not obtain a significant *p-value* in any of the tests, which means we are not able to accept the hypothesis that the experience influences their perceptions on which challenges they face.

Therefore, given the results we obtained, we conclude that experience does not play an important role in the results we presented throughout this paper.

6 Threats to Validity

In this section, we discuss the threats to the validity of this study as well as the actions we took to mitigate them. We divide this section according to the three different methodologies we apply in this study.

6.1 Part I. Interviewing JavaScript Developers

Transferability. The main limitation to this part of the study is its possible lack of generalizability. The sample size is not large and it thus may not represent all OSS development. Moreover, as we only talk to developers from OSS projects, the results may not represent industry software. We tried to mitigate this fact by interviewing experienced developers from popular and reputable projects. However, that selection of the sample creates another bias in the study where

the results might be different if smaller projects were examined along with projects having smaller configurations. Only projects were selected that had somewhat extensive configurations as we then expected to receive more input on how rules are selected for a project, thus being able to report on what methods are used for the task.

As we only looked at projects that use ESLint, the results might not reflect on usage of all JavaScript linters. Also examining other linters such as JSLint or JSHint might produce different results than presented in this study, which would be an interesting aspect to see in future studies. We chose to observe the usage of only one linter to make the interviews more consistent. As the available linters have different features, *e.g.*, regarding configurability, we would not have been able to ask all participants the same questions, and thus possibly making the analysis less reliable. To minimize the effects of this possible lack of transferability, we chose to address the most popular and most flexible linter, also to not restrict the results with more limited use cases. Additionally, we verified that ESLint is indeed the most popular linter among the top 120 JavaScript projects by manually examining the linter configurations of each project.

Credibility. Possible variables that effect the results of this study relate to the previous knowledge of the participants. It is likely that we interviewed people who already feel strongly about linters as they are frequent users of the tool. We can not know for sure if their opinions are based on their own experience with using the tool or if it is based on external literature that they have read. Because of this concern, we tried to address our questions to relate specifically to the participants' own opinions and experience working on the particular project. However, in some cases, the participants had other and even more experience in working with a linter on other projects, for which they also based their answers on.

Confirmability. A possible limitation concerns the analysis of the interviews. The interviews were conducted by the first author only, who also analyzed the transcripts. In order to reduce any possible bias that this decision could bring to the results, the first and the second authors of this paper met after every conducted interview and discussed about what that new piece of information brought to the overall results. Any existing conflict of ideas were solved in these meetings. Both researchers also used these meetings to refine the final codes that were used as main topics throughout the Results section of this paper.

Due to this iterative process, we did not calculate inter-rater agreement, as commonly done in qualitative studies. Nevertheless, it is possible that someone else would have constructed the codes differently, perhaps resulting in different conclusions [70]. We therefore make the derived codes available online for inspection [35].

6.2 Part II. Mining Linter Configurations in Open Source Systems

Sample Selection. Our sample selection procedure consisted of selecting all JavaScript projects on GitHub with more than 10 stars. As we explain in the Methodology section, our goal was to only analyze repositories that can possibly count as "real" software projects, *i.e.*, not short lived coding experiments or small personal projects. Although it is well known that stars on GitHub do not necessarily represent the quality (nor the size) of a project [71], manually selecting them would have been costly and possibly biased. Nevertheless, as we see in Section 3.2.3, our sample contains open source projects that highly vary in terms of code size and creation dates. Future work should focus on replicating this study in industry projects.

Reliability of the Linter Detection Strategy. A limitation of the tool is that it can possibly miss out on some ESLint configuration files. The configuration file is typically located in the main directory of a project (as it will then be used for the whole project), so in order to save execution time and to simplify the tool, it is the only location where the tool searches for the file. It could however be the case with some projects that they place their configuration file in a subdirectory of the project and pass it to the linter with the command line. This limitation does however not have any significant effects on the results, since the only real implication is that the dataset for analyzing the configuration files is smaller than it could have been. A trivial limitation is that the numbers are skewed regarding the prevalence of using ESLint, as reported in Section 3.2.3 about the dataset's characteristics. That information is however not of main interest for this study.

Continuing on the topic of the prevalence of linter usage, there are limitations in acknowledging the presence of other linters as well. As the tool does not check for all linters that exist, no assumptions can be made about the overall prevalence of linter usage. For example, the tool does not account for usage of JSLint, but it was decided to disregard that linter since the usage is less frequent than of the other linters and it was more difficult to identify its usage.

Moreover, for some linters it is not necessary to have a configuration file in place and it is also possible to specify a configuration file with other names than the default recognized formats. For these reasons the usage could as well be understated. On the other hand, the usage in some cases can be overstated. Even though a project has a configuration file for a specific linter, it does not guarantee that the linter is actually used. As an example, the tool found that one project has four linters enabled, having configuration files for ESLint, JSHint and JSCS along with having Standard as a dependency [72]. On closer inspection one can see that only one of them seems to be currently in use (and even additionally uses TSLint that is intended for TypeScript).

Further measures could have been taken to make this analysis more accurate, such as analyzing the dependencies of all projects along with all its pre-running scripts. That would however still not guarantee true results and was considered out of scope, as this is not the main purpose of the study. For these reasons, the prevalence results are presented as estimations of the usage and not as definite findings.

To increase confidence in the current implementation to identify the usage of ESLint, JSHint, JSCS and Standard, manual and automated unit testing were applied. Actual use cases of all covered situations to detect these linters were used as input to see if the tool identified them correctly.

Data Analysis. Our linter detection tool also took the

opportunity to summarize the data on-demand (*i.e.*, count the number of projects that do and do not use presets, number of enabled and disabled rules, as well as the arithmetic mean of rules per project), as we show in Table 8. Our tool did not store the raw files for future analysis. As we only report the arithmetic mean, readers do not have the opportunity to explore the entire distribution, which could lead them to different insights. In future work, we plan to better understand the distribution of the enabled, disabled, and warned rules.

6.3 Part III. Surveying Developers

Survey Design and Analysis. It is important to consider both the *validity* and *reliability* of a survey [45], [46]. The validity mainly has to do with how well surveys measure what they are intend to measure, while reliability considers how consistent and true the responses are. To try to ensure that this survey was both valid and reliable, much consideration went into writing relevant questions that use direct and appropriate language. All questions were then reviewed and evaluated in pilot tests with members of the target population. To further evaluate the reliability of the survey, the responses were reviewed in separate batches based on the source of the participants, *e.g.*, a separate batch for all participants that entered the survey via the promotion on Reddit. We did not observe any major discrepancies between the different groups.

For the coding of the open questions, we followed a strategy similar to Part I, where the first author performed the initial coding, and then refined the results by means of several discussions with the second author. Similarly to the threats of Part I (Section 6.1), it is however possible that someone else would have coded the responses differently, perhaps resulting in different conclusions [70]. We therefore also make the derived codes available online for inspection [35].

Survey Evaluation. The survey was pilot tested in order to identify possible problems and to make general improvements. Six participants were recruited who provided valuable feedback for the survey where some crucial improvements were made. After these six tests, the authors were confident that these improvements were sufficient for the survey to be considered both valid and reliable.

It is however possible that not all problems were detected in these tests and that more tests would have led to making more improvements. To make the pilot tests as efficient and effective as possible, several sources of literature were examined to learn about what aspects to focus on. This preparation played a key part in the resulting confidence that the authors had about the final quality of the survey. Additionally, when examining the final responses of the survey, no major problems could be identified with the answers, such as anything implying a wide misunderstanding of any questions.

Sampling and Responses. It is important to consider the response rate and the representativeness of a survey's response set [73], but it is however difficult to calculate a response rate for this kind of convenience sampling. For two of the distribution places, JS.is and Echo JS, it is impossible to know how many individuals viewed the original posts to estimate a response rate. For JS Reddit, it is known that the post was opened by 1,300 individuals, resulting in a response rate of 20.6% for complete responses[17]. It is however not known how many people saw the post on the front page and did not decide to take a further look, consequently not having access to the survey link. For the last location, Twitter, the tweet reached a number of 4,615 users but it is difficult to predict how many of those users actually noticed or read the tweet.

Perhaps more important in this case is the representativeness of the response set. Three of the locations (JS.is, JS Reddit and Echo JS) are communities that are specifically created for JavaScript enthusiasts, which is exactly the target group of the study. Furthermore, as the title and description of the survey refer to linters for JavaScript, it should attract mostly JavaScript developers that have used a linter in the past. In the case of Twitter, all retweeters except for one (who did not have a self-description) are either developers or software engineering professors or researchers. These people most likely have many followers of related professions where however not all might be involved with JavaScript.

We observe that JavaScript is the main language for 96.1% of the participants, with an average experience of 5.8 years with using JavaScript and every single respondent had used the language in the last 12 months. Furthermore, 87.0% regularly work in industry and 50.6% with OSS, which allows this analysis to cover both the commercial and open source sectors. Additionally, the sample is very diverse regarding the participants' country of residence, leading to the results not representing only one market or one part of the world.

However, there is almost no gender diversity in the sample, which is unfortunately a common problem when performing surveys within the software engineering field [74]. Even companies like Stack Overflow, with high media impact, have trouble with gender balance in their popular surveys (in the 2018 survey with more than 100k respondents, only 6.9% identified themselves as female [75]). Studies have shown that there exists bias against women in open source communities [76] and the authors of this paper are strongly committed to gender diversity. Future work must explore different JavaScript communities with a better gender balance.

7 Related Work

Static analysis has been a popular research topic where many tools have been created, both for academic and industrial use. These tools can be convenient as they are used without executing the software which they are applied on. A common research topic is the construction of such tools, such as for security vulnerability, fault detection, and code smells [77], [78], [79], [80], [81], [82], and the various methods to do so, such as data flow, information flow, path or pointer analysis [5]. Other research topics explore the usage of these tools, which we are more interested in for this particular study. These topics include developers' perceptions of static analysis tools [6], [8], the value of using these tools in real world applications [55], [5] and the

17. It is estimated that 378 partial responses were received from Reddit, resulting in a response rate of 49.7% for all responses.

challenge of false positives [10], [56]. In the following we briefly explore these latter topics and finally discuss static analysis in the world of JavaScript.

7.1 Perceptions of Static Analysis Tools

Johnson *et al.* [8] researched the usage of static analysis tools, in particular why some developers do not use these tools to find bugs despite of their proven benefits. They conducted a study where they interactively interviewed 20 participants while applying FindBugs on software. They found that the main reasons why the participants chose to use static analysis tools were to avoid the time consuming manual labor to find bugs, to support team development efforts and to enforce coding standards. Reasons to not use these tools include poorly presented output where there are too many false positives or too many warnings reported in general, in addition to tools not being integrated conveniently in the workflow. Moreover, most participants claimed that it is important for these tools to be easily customizable and that warnings should be explained properly, including how they can be fixed. Our work is inspired by this study where we also research why static analysis tools are used and how they can be improved, but focusing solely on JavaScript. We see that the overall expectation of JavaScript developers with static analysis tools are similar to the ones in previous research.

A similar study was conducted by Christakis and Bird [6] who also investigate how developers perceive static analysis tools, specifically which barriers they face in the adoption of these tools and how these tools should be created so that developers will take advantage of them. For this purpose they surveyed a random sample of 375 Microsoft developers. What was considered to be the largest obstacle in using these tools was the fact that some unwanted rules are turned on by default in the tools' configurations. Proposed solutions to this problem are to have a subset of rules enabled instead of all rules being enabled by default, or to make the configuration process easy for developers. This idea is similar to what ESLint does; there are no default configurations but one can easily enable a preset with a selected subset of rules that the creators of ESLint think are the most important (*eslint:recommended*). In addition, in RQ_2, we explore the different strategies that JavaScript developers have been applying in order to derive the best configuration for their projects. We conjecture our findings also make sense for static analysis tools of other languages.

Moreover, other frequently experienced challenges, as reported by Christakis and Bird, were bad warning messages, too many false positives and for the analysis to be too slow. For the functionality that these developers want static analysis tools to detect, security issues, best practices and concurrency issues were the most commonly requested ones. Issues relating to style were among the least requested features for such a tool to have. However, the majority of the developers reported that the issues they encounter that can be caught by static analysis tools are most commonly best practices violations and style inconsistencies, opposed to more complex issues such as regarding reliability or concurrency. The expectations of these developers therefore seem to match with a tool like ESLint. Our results show that, although developers agree that *Possible Errors* is a highly important category to be enabled, in practice, only 52% of projects actually enable such rules. Our hypothesis is somewhat inline with previous research, where we conjecture that the set of available *Possible Errors* rules is not sufficient. Thus, more research needs to be conducted in order to provide developers with a set of rules that are able to detect more complex and important bugs.

7.2 Configurations of Static Analysis Tools

Ayewah *et al.* [7], [83] studied the usage of FindBugs where they saw that users are generally interested in fixing warnings from the tool, especially the high priority ones, but which types of warnings depends on the user's context. It is therefore deemed valuable to have configuration options for these different groups of users. Similarly, Jaspan *et al.* [2] recognized the importance of customizing and prioritizing rules to make developers more willing to use a static analysis tool as it reduces the number of perceived false positives for a project.

Regarding how developers configure these tools, Beller *et al.* [3] performed a large-scale study on the prevalence of static analysis tool usage along with how they are configured. They examined static analysis tools for four languages, Java, JavaScript, Python and Ruby, where for JavaScript they observed JSHint, ESLint, JSCS and JSL [84]. They found that these tools are commonly used in OSS software (over half of all analyzed projects) and in particular for JavaScript projects where JSHint was by far the most widely used linter[18]. The configurations for these projects are most often changed from the default settings, but typically only one rule is added, removed or modified. Furthermore, after a configuration file has been created, they are rarely modified. For the types of rules that are configured, those that are related to maintainability were both more commonly enabled and disabled than those that have to do with functional defects. This is in line with our findings as we have seen that rules from the categories *Stylistic Issues* and *Best Practices* are enabled and disabled more often than rules from the *Possible Errors* category. Beller *et al.* predict that fewer bug-finding rules are enabled as tools are known to perform poorly in finding defects.

Zampetti *et al.* [9] further analyzed the usage of static analysis tools in continuous integration (CI) pipelines in popular OSS projects. For the 20 analyzed projects, Checkstyle was the most commonly used static analysis tool, followed by FindBugs, where 11 projects used only one tool while the rest employed multiple tools. Most of these static analysis tools were configured to break the build in the CI pipeline, especially in the case of Checkstyle (only one project solely raised warnings). Most projects manually configured the rules that were activated in these projects for at least one tool that was used, where Checkstyle was configured in every single case. While FindBugs was typically configured to include all available rules, the percentage of used rules for Checkstyle always remained below 40%. The most commonly enabled Checkstyle rules were regarding indentation (*Whitespace*), unused imports (*Imports*), possible defects (*Coding*) and code blocks (*Block*

18. At the time of analysis in early 2015.

checks). The configurations of the static analysis tools were not changed often over the projects' lifetime, each with less than 10 modifications except for one project. Zampetti *et al.* recommend developers to configure static analysis tools to their needs to benefit the most from using these tools, *e.g.*, to avoid unnecessary build failures.

Some of the previous literature indeed included ESLint but we suspect that these results would be somewhat different if these studies would be conducted again today, as ESLint has since then removed any default configurations in the tool and all rules are now off by default. A developer using the tool thus now needs to create some configuration, which in the simplest case could just include the recommended settings from ESLint.

In addition, the related work does not perform a fine-grained analysis on the rules that are enabled and disabled, as we proposed. Our results bring clear understanding on the prevalence of each existing rule in ESLint. We suggest researchers to replicate our approach in the configuration of static analysis tools in other languages.

7.3 Effectiveness of Static Analysis Tools

Several studies have focused on the effectiveness of static analysis tools to find bugs in software systems. In a case study, Wagner *et al.* [85] found that static analysis tools report on different warnings than those that are found by testing a software, indicating that it is beneficial to use these two methods in combination. Furthermore, static analysis tools find a subset of the issues that are found via code review, thus if used beforehand, they can save some of the time that goes into performing the manual task. Zheng *et al.* [5] also found that static analysis tools are complementary to other fault-detection techniques but with limited usability where testing was found to be more effective.

Wedyan *et al.* [55] further examined how effective static analysis tools are in detecting faults and refactoring opportunities. Studying two software systems, three known static analysis tools were only able to detect 3% of the defects that were actually found in the projects' lifetime. These tools were however much more successful in identifying the refactorings that were performed in the project, or around 70%. Similarly, Couto *et al.* [61] also found that FindBugs was not able to identify most of the errors that had been fixed in some studied software systems. However they found some level of correlation between the number of reported warnings and the number of actual field bugs. So despite the fact that static analysis tools may not be good at finding actual field defects, they can serve as good indicators of the level of internal quality in a software system.

In our paper, although we extensively report how developers have been using linters, which rules they opt to enable and disable, as well as their perceptions on how important each of them are, we do not measure how effective each rule really is. In future work, we intend to understand and measure the quality of these rules.

7.4 False Positives in Static Analysis Tools

A common problem with static analysis tools is the high volume of false positives [55], [8], [10], [56]. For example, Wedyan *et al.* [55] found well known static analysis tools to produce a false positive rate above 90% when applying them on several software systems. The presence of false positives is indeed known to be one of the main barriers of static analysis tool adoption [8], [6], [4]. Developers will not tolerate a high false positive rate and quickly discard a tool if the rate is too high [6], [65]. More specifically, Christakis and Bird [6] found that most developers do not tolerate a false positive rate of over 20%. They prefer for tools to identify fewer defects, rather than catching more bugs and having a higher volume of false positives. Furthermore, Wagner *et al.* [85] experienced that when static analysis tools report a high number of false positives, it can take even more time to shift through them to find the true defects, than the time that is saved on manual effort by using a static analysis tool in the first place.

The term false positive can be understood in two different ways, either a wrongly reported warning or a true warning that is not considered by a developer to improve the software under analysis [11]. Tool makers have thus experienced that the users decide for themselves what a false positive is and therefore how effective a tool is [65]. Additionally, when tools identify more intricate issues, the risk increases of users not understanding the issue and mistaking true positives for false positives [10]. In general when there are many reported false positives, developers start trusting the tool less and less [10]. With less trust, even more true positives are thought to be false positives by the developers. Couto *et al.* [61] and Jaspan *et al.* [2] further experienced that developers categorize warnings that are not of their interest as false positives. They stressed the importance of customizing tools and prioritizing rules so that only high-priority and relevant warnings are reported. Without doing so, tools can not become practically usable.

As we discuss in Section 5.3.2, although our participants see false positives as a challenge, they perceive them to occur infrequently. We raised a few hypotheses on why this happens in that section. Therefore, we invite researchers to conduct studies on the prevalence of false positives in the existing JavaScript linters.

7.5 Static Analysis Tools Used in Industry

An abundance of static analysis tools have been developed with diverse features and implementations, where software systems have different requirements for these tools. Many static analysis tools have been used by major software companies such as Google, Facebook and Microsoft, both tools that have been created internally and externally [6]. Bessay *et al.* [10], the creators of Coverity, experienced first hand how difficult it can be to create a static analysis tool that fits for commercial use of companies with large code bases. The technical requirements of such companies can vary greatly, *e.g.*, using miscellaneous build processes and developing environments that can make running the tool different for each particular case. The size and nature of software projects can also entail different requirements for static analysis tools, where Google resorted to creating their own tool as no other available static analysis tool could handle the size of their codebase [65].

Google have indeed tried and reported from using several static analysis tools such as FindBugs and Coverity [86].

Despite some success with adopting these tools, ultimately they were not frequently used by developers due to the presence of false positives, lack of scalability and inconvenient workflow integration [65]. Eventually they instead created their own tool, Tricorder [65], a static analysis tool with an ecosystem of developers creating and reviewing custom analyzers. They only display high priority warnings from analyzers that have received good reviews from other developers, in an attempt to make the developers more willing to use the tool by eliminating false positives and low priority warnings.

EBay [2] reported on their substantial effort into choosing the most valuable and appropriate tool set for their projects. They report on a method to evaluate these tools where eventually the tool FindBugs was integrated to their development process. For their software it was deemed important that the adopted tool should be extensible (to allow for project custom rules) and customizable. They report that customization and prioritization are important steps in evaluating a tool as it can reduce the false positive rate for a project and make developers more willing to use the tool.

In our paper, although some of the interviewees were also developers in industry, the extensive mining analysis we performed was conducted solely on open source systems. Future work should focus on how companies are configuring such linters.

7.6 Static and Dynamic Analysis for JavaScript

As JavaScript has become more popular it has also attracted more attention in the research community. The need for more and better tooling for the language has been emphasized along with the challenges that JavaScript introduces for proper static analysis. Several studies have therefore focused on techniques for, and the creation of, static analysis tools for JavaScript [87], [88], [89].

JavaScript is a dynamic language, which *e.g.*, involves dynamic typing, dynamic file loading, first-class functions and property access, and in general code that can be generated during runtime. Richards *et al.* [57] analyzed the usage of dynamic features in JavaScript code in commonly used websites and found that some dynamic features are indeed frequently used, such as adding and removing properties to objects after their initialization and using the *eval* function for various unpredictable purposes. As such, JavaScript has been described as harsh terrain for static analysis [57], [89], [90], [91]. The usability of static analysis tools for JavaScript has thus also been criticized and said to be limited since these tools can not account for runtime behavior [58]. For this problem, tools have been created that employ dynamic analysis of program executions [91], [58], [92]. A recent example is DLint [58] that, evaluated against JSHint, detected on average 49 new code quality issues per studied system. Example issues they identified are object properties shadowing prototype properties, passing too many arguments to a function, accessing the *undefined* property, and silent coercion of arithmetic operations to *NaN*. Some issues were detected by both DLint and JSHint but with different methods to do so, where DLint increased the total warnings found by 10%, thus complementing JSHint.

In general, while there have been several proposed static analysis tools and techniques for JavaScript in previous research, most suffer from being *sound* but impractical, or the opposite, practical but *unsound* [93], [91]. Practical limitations include lack of scalability and inabilities to analyze library usage. More tools have been developed for specific challenges of JavaScript, such as the prevalent use of external libraries [94], [89]. Some large libraries, such as the Browser API and HTML DOM, are written in other languages and do not provide a JavaScript implementation, thus making static analysis more difficult.

Other common libraries like jQuery [95] are also known to make extensive use of many dynamic language features, such as use of *eval* and computed property names, which also contributes to the difficulty of static analysis for JavaScript code. Another related challenge is the dynamic file loading of these libraries where the code is therefore not made available immediately for analysis. For this problem, researchers have proposed *staged* analysis [96], [97] which is applied in two stages, first for all statically available code and later for the remaining dynamically loaded code. Interestingly, the false positives that have been identified with static analysis of JavaScript code (by applying the tool SAFE [98]), mostly had to do with API usage and asynchronous function calls, though also due to dynamic file loading and dynamic code generation [99].

Despite the challenges that it brings, automated static analysis can be especially beneficial for languages that do not have strict typing [5]. Type analysis is considered to be a crucial part to catch representation errors where, for example, numbers can be confused with strings or functions confused with booleans [87], [15]. Additionally, simple mistakes such as spelling mistakes can result in surprising consequences at run time, which is mostly avoided in other languages with static typing [90]. Pradel *et al.* [15] created a tool, TypeDevil, to counter against errors that have to do with inconsistent types in JavaScript, using dynamic analysis where types of variables, properties and functions are observed at runtime. Moreover, as typing systems have been developed such as TypeScript [100], Gao *et al.* [101] studied the effects of dynamic typing on errors in JavaScript applications and examined whether typing systems could prevent them. They added type annotations to code that had caused errors in software systems and examined whether using TypeScript or Flow on that code would surface the previously encountered errors. They saw that applying these static type systems would detect many errors (15% in this study) that otherwise can go unnoticed into production code.

As our results show, developers acknowledge the fact that JavaScript is a dynamic language, and thus, they accept the fact that static analysis tools may have a hard time in analyzing dynamic behavior. However, developers also showed their interest in seeing such analysis. Our findings reinforce the importance of more research in the field of static and dynamic analysis of dynamic languages.

Lastly, code smells in JavaScript have recently been studied where Fard *et al.* [92] developed a tool, JSNose, to detect potential smells. Saboury *et al.* [102] later analyzed the prevalence and impact of several code smells and surveyed developers on their perceptions of those smells. The studied

code smells were derived from literature and online style guides, including recommendations from ESLint and from the Airbnb preset. They found that JavaScript files that are affected with code smells tend to be more error-prone than those without any smells. Moreover, *Variable Re-assign* and *Assignment in Conditional Statements* were among the most dangerous smells. Additionally, when surveying a large sample of JavaScript developers, *Nested Callbacks*, *Variable Re-assign* and *Long Parameter List* were perceived to be the most hazardous smells affecting the maintainability and reliability of JavaScript applications.

8 Conclusion

In this study we have applied three research methods where we first interviewed 15 experts on using ESLint, then analyzed over 9,500 ESLint configuration files from GitHub projects and finally surveyed more than 300 JavaScript developers about their experiences with using a linter, exploring the findings from the previous two analyses.

Our findings can be summarized as follows:

- We see that most developers use a linter to maintain code consistency. Nevertheless, both interview and survey participants claim that rules that relate to possible errors are far more important than those that relate to stylistic issues.
- Regarding how developers configure linters, they generally prefer to use a preset rather than to manually choose every rule that is used for a project. However, most developers additionally apply some configurations that are specific to a project or a team.
- While it is important to be able to identify errors early in the development process by using a linter, even more developers feel the need to have consistent formatting where everyone taking part in a project uses quotes, semicolons and indentation in the same way.
- Upholding these configurations can be a challenging task where it can be difficult to agree with teammates on which rules should be used for a project.

The results of this study produced valuable implications for developers, tool makers and researchers, who can use the information to make better use of linters, to improve future versions of linters and to further research this important aspect of software development.

References

[1] B. W. Boehm, *Software engineering economics*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 197.

[2] C. Jaspan, I. Chen, A. Sharma *et al.*, "Understanding the value of program analysis tools," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 963–970.

[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 *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 470–481.

[4] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. IEEE, 2004, pp. 245–256.

[5] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE transactions on software engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[6] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 332–343.

[7] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, 2008.

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[9] 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 *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 334–344.

[10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[11] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 1–8.

[12] GitHub, "Language Trends on GitHub," https://github.com/blog/2047-language-trends-on-github, [Online; accessed 13-June-2017].

[13] F. S. Ocariza Jr, K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*. IEEE, 2011, pp. 100–109.

[14] T. Mikkonen and A. Taivalsaari, "Using JavaScript as a real programming language," *Sun Microsystems, Inc.*, 2007.

[15] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 314–324.

[16] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[17] "ESLint," http://eslint.org, [Online; accessed 11-July-2017].

[18] P. Vorbach, "npm-stat, download statistics for packages eslint, jshint, jslint, jscs, standard," https://npm-stat.com/charts.html?package=eslint&package=jshint&package=jslint&package=jscs&package=standard&from=2015-01-01&to=2017-05-31, [Online; accessed 2-June-2017].

[19] B. G. Glaser and J. Holton, "Remodeling grounded theory," in *Forum Qualitative Sozialforschung/Forum: Qualitative Social Research*, vol. 5, no. 2, 2004.

[20] K. F. Tómasdóttir, M. Aniche, and A. v. Deursen, "Why and how javascript developers use linters," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 578–589.

[21] "FindBugs," http://findbugs.sourceforge.net, [Online; accessed 11-July-2017].

[22] "Checkstyle," http://checkstyle.sourceforge.net, [Online; accessed 11-July-2017].

[23] "PMD," https://pmd.github.io, [Online; accessed 11-July-2017].

[24] "JSHint," http://jshint.com, [Online; accessed 11-July-2017].

[25] "JSCS," http://jscs.info, [Online; accessed 11-July-2017].

[26] "JSLint," http://jslint.com, [Online; accessed 11-July-2017].

[27] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2011, pp. 52–78.

[28] ESLint, "Rules," http://eslint.org/docs/rules, [Online; accessed 2-June-2017].

[29] AirBnb ESLint preset. [Online]. Available: https://github.com/airbnb/javascript

[30] "Standard JS," https://standardjs.com, [Online; accessed 11-July-2017].

[31] ESLint, "Configuring ESLint," http://eslint.org/docs/user-guide/configuring, [Online; accessed 27-May-2017].

[32] S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *Empirical Software Engineering*, vol. 16, no. 4, pp. 487–513, 2011.

[33] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 120–131.

[34] S. E. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *Software metrics, 2005. 11th ieee international symposium*. IEEE, 2005, pp. 10–pp.

[35] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, "The Adoption of JavaScript Linters in Practice: A Case Study on ESLint [Data set]," https://doi.org/10.5281/zenodo.1410967, zenodo.

[36] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[37] GitHub, "About Stars," https://help.github.com/articles/about-stars, [Online; accessed 6-March-2017].

[38] "BigQuery," https://cloud.google.com/bigquery, [Online; accessed 29-May-2017].

[39] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132

[40] "GHTorrent," http://ghtorrent.org, [Online; accessed 29-May-2017].

[41] "ESLint first release (v0.0.2)," https://github.com/eslint/eslint/releases?after=v0.1.0, [Online; accessed 23-May-2017].

[42] "ESLint first major release (v1.0.0)," https://github.com/eslint/eslint/releases?after=v1.0.0, [Online; accessed 23-May-2017].

[43] "stream-handbook via GitHub (example of a repository containing a programming guide)," https://github.com/substack/stream-handbook, [Online; accessed 23-May-2017].

[44] "Impress.js-Tutorial via GitHub (example of a repository containing code for a tutorial)," https://github.com/cubewebsites/Impress.js-Tutorial, [Online; accessed 23-May-2017].

[45] A. Fink, *The survey handbook*. Sage, 2003, vol. 1.

[46] D. De Vaus, *Surveys in social research*. Routledge, 2013.

[47] B. A. Kitchenham and S. L. Pfleeger, "Principles of survey research: part 1: turning lemons into lemonade," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 6, pp. 16–18, 2001.

[48] K. F. Tómasdóttir, "Post promoting survey on JavaScript user group (Iceland) on Facebook," https://facebook.com/groups/nodejsis/permalink/1709121425832578, [Online; accessed 14-June-2017].

[49] ——, "Post promoting survey on Reddit." https://redd.it/6eumsp, [Online; accessed 14-June-2017].

[50] "Echo JS," http://echojs.com, [Online; accessed 14-June-2017].

[51] K. F. Tómasdóttir, "Post promoting survey on Twitter." https://twitter.com/kristinfjolato/status/871986432952479747, [Online; accessed 14-June-2017].

[52] "eslint-config-airbnb via npm," https://npmjs.com/package/eslint-config-airbnb, [Online; accessed 29-May-2017].

[53] "eslint-config-airbnb-base via npm," https://npmjs.com/package/eslint-config-airbnb-base, [Online; accessed 29-May-2017].

[54] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 41–50.

[55] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. IEEE, 2009, pp. 141–150.

[56] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[57] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 1–12.

[58] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: Dynamically checking bad coding practices in JavaScript," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 94–105.

[59] "TypeScript," http://typescriptlang.org, [Online; accessed 10-July-2017].

[60] "Flow," http://flowtype.org, [Online; accessed 10-July-2017].

[61] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente, "Static correspondence and correlation between field defects and warnings reported by a bug finding tool," *Software Quality Journal*, vol. 21, no. 2, pp. 241–257, 2013.

[62] "AirBnb ESLint base preset," https://github.com/airbnb/javascript/tree/master/packages/eslint-config-airbnb-base, [Online; accessed 15-January-2018].

[63] "Google ESLint preset," https://github.com/google/eslint-config-google, [Online; accessed 15-January-2018].

[64] "Standard ESLint preset," https://github.com/standard/eslint-config-standard, [Online; accessed 15-January-2018].

[65] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 598–608.

[66] W. Fong Boh, S. A. Slaughter, and J. A. Espinosa, "Learning from experience in software development: A multilevel analysis," *Management Science*, vol. 53, no. 8, pp. 1315–1331, 2007.

[67] A. Beer and R. Ramler, "The role of experience in software testing practice," in *Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference*. IEEE, 2008, pp. 258–265.

[68] P. E. Greenwood and M. S. Nikulin, *A guide to chi-squared testing*. John Wiley & Sons, 1996, vol. 280.

[69] T. V. Perneger, "What's wrong with bonferroni adjustments," *BMJ: British Medical Journal*, vol. 316, no. 7139, p. 1236, 1998.

[70] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[71] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[72] "MQTT.js via GitHub (example of a repository containing four linters)," https://github.com/mqttjs/MQTT.js, [Online; accessed 26-May-2017].

[73] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[74] B. A. Kitchenham and S. L. Pfleeger, "Principles of survey research: part 5: populations and samples," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 5, pp. 17–20, 2002.

[75] S. Overflow, "Stack Overlow Developer Survey 2018." https://insights.stackoverflow.com/survey/2018/, [Online; accessed April-2018].

[76] J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. Murphy-Hill, C. Parnin, and J. Stallings, "Gender differences and bias in open source: Pull request acceptance of women versus men," *PeerJ Computer Science*, vol. 3, p. e111, 2017.

[77] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis." in *USENIX Security Symposium*, vol. 14, 2005, pp. 18–18.

[78] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," Wisconsin Univ-Madison Dept of Computer Sciences, Tech. Rep., 2006.

[79] M. Aniche, G. Bavota, C. Treude, A. Van Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architectures," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 233–243.

[80] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, "Satt: Tailoring code metric thresholds for different software architectures," in *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE, 2016, pp. 41–50.

[81] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, pp. 1–37, 2017.

[82] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 2008, pp. 329–331.

[83] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 workshop on Defects in large software systems*. ACM, 2008, pp. 1–5.

[84] "JavaScript Lint," http://javascriptlint.com, [Online; accessed 7-August-2017].

[85] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing bug finding tools with reviews and tests," *Lecture Notes in Computer Science*, vol. 3502, pp. 40–55, 2005.

[86] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 241–252.

[87] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *SAS*, vol. 9. Springer, 2009, pp. 238–255.

[88] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of JavaScript," *European Conference on Object-Oriented Programming (ECOOP)*, pp. 435–458, 2012.

[89] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of JavaScript applications in the presence of frameworks and libraries," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 499–509.

[90] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript web applications," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 59–69.

[91] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 165–174.

[92] A. M. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript code smells," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013, pp. 116–125.

[93] Y. Ko, H. Lee, J. Dolby, and S. Ryu, "Practically tunable static analysis framework for large-scale JavaScript applications (T)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 541–551.

[94] E. Andreasen and A. Møller, "Determinacy in static analysis for jQuery," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 17–31.

[95] "jQuery," https://jquery.com, [Online; accessed 13-June-2017].

[96] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 50–62, 2009.

[97] S. Guarnieri and B. Livshits, "GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications." *WebApps*, vol. 10, pp. 6–6, 2010.

[98] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, vol. 10, 2012.

[99] J. Park, I. Lim, and S. Ryu, "Battles with false positives in static analysis of JavaScript web applications in the wild," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 61–70.

[100] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *European Conference on Object-Oriented Programming*. Springer, 2014, pp. 257–281.

[101] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: quantifying detectable bugs in JavaScript," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 758–769.

[102] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in JavaScript projects," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 2017, pp. 294–305.

Kristín Fjóla Tómasdóttir is a Software Developer at Amazon in Cambridge, United Kingdom. Kristín completed her (cum laude) MSc degree at Delft University of Technology, The Netherlands. Her research focused on an empirical evaluation of JavaScript linters in real-world open source systems.

Maurício Aniche is an Assistant Professor at Delft University of Technology, The Netherlands. Maurício helps developers to effectively maintain, test, and evolve their software systems. His current research interests are systems monitoring and DevOps, empirical software engineering, and software testing.

Arie van Deursen is professor in software engineering at Delft University of Technology, The Netherlands, where he heads the Software Engineering Research Group (SERG) and chairs the Department of Software Technology. His research interests include empirical software engineering, software testing, and software architecture. He aims at conducting research that will impact software engineering practice, and has co-founded two spin-off companies from earlier research. He serves on the editorial boards of Empirical Software Engineering, and the open access PeerJ/CS.