Exploring API Deprecation Evolution in JavaScript

Romulo Nascimento, Andre Hora, Eduardo Figueiredo
Department of Computer Science
Federal University of Minas Gerais, Brazil
{romulonascimento, andrehora, figueiredo}@dcc.ufmg.br

***Abstract*—Building an application using third-party libraries is a common practice in software development. As any other system, software libraries and their APIs evolve. To support version migration and ensure backward compatibility, a recommended practice during development is to deprecate API. Different from other popular programming languages such as Java and C#, JavaScript has no native support to deprecate API elements. However, several strategies are commonly adopted to communicate that an API should be avoided, such as the project documentation, JSDoc annotation, code comment, console message, and deprecation utility. Indeed, there have been many studies on deprecation strategies and evolution mostly on Java, C#, and Python. However, to the best of our knowledge, there are no detailed studies aiming at analyzing how API deprecation changes over time in the JavaScript ecosystem. This paper provides an empirical study on how API deprecation evolves in JavaScript by analyzing 1,918 releases of 50 popular packages. Results show that close to 60% have rising trends in the number of deprecated APIs, while only 9.4% indicate a downward trend. Also, most deprecation occurrences are both added and removed on minor releases instead of removed on major releases, as recommended by best practices.**

***Index Terms*—API deprecation, JavaScript library, Software evolution, Software maintenance, Software mining**

I. Introduction

Developing an application using third-party libraries and frameworks is a common practice nowadays. Libraries provide reusable functionality to client applications via Application Programming Interfaces (APIs). API usage brings several advantages to a software development project [1], such as cost and resource usage reduction. As a result, developers can focus on business core requirements since software quality may increase by relying on libraries that have been widely adopted, tested and documented [2].

As any other software system, libraries, frameworks and their APIs evolve [3]. Thus, functions and parameters might be renamed, updated, moved, or removed. Consequently, client applications are encouraged to migrate to the latest stable versions of their dependencies [4] to benefit from novel APIs, security improvements, and bug fixes. To help version migration and ensure backward compatibility, a recommended practice in software development is to deprecate the API. In other words, deprecation indicates that the use of a certain API should be avoided because it will be changed, removed or discontinued in a future version [5]. Some of the most popular programming languages, such as Java and C#, provide native support mechanisms and tools to help developers explicitly deprecate their APIs [6]. Although JavaScript has no native support to deprecate API elements, several strategies are commonly adopted to communicate that the use of an API should be avoided, such as the project documentation, JSDoc annotation, code comment, console message, and deprecation utility [7].

As code evolves, deprecated APIs need to be maintained as well. When deprecation is needed, JavaScript developers suggest that it should be added on minor releases, and then removed on a later major version, in which breaking changes are usually introduced [7]. However, to the best of our knowledge, there are no detailed studies aiming at analyzing how API deprecation evolves over time in the JavaScript ecosystem, and whether those recommendations are actually followed.

JavaScript has become extreme popular over the last years,[1] and so has the usage of third-party packages. Therefore, this motivates us to investigate how deprecated APIs evolve over the lifetime of third-party JavaScript packages. In particular, we analyze (i) trends in the number of deprecated APIs in 50 popular JavaScript packages, and (ii) the type of releases in which deprecated APIs are usually added and removed. Thus, we proposed the following two research questions:

- *RQ1: How do deprecated APIs evolve in JavaScript packages?*
- *RQ2: To what extent are deprecated APIs introduced and removed in release types?*

We mine the source code of 1,918 versions from the 50 most-dependent upon JavaScript packages, according to *Libraries.io*, and analyze how the number of deprecated APIs increase and decrease between version releases. Our results suggest that close to 60% of the projects present an increase in the number of deprecated APIs, while only 9.4% indicate decreasing trends. Additionally, deprecation is usually added and removed on minor releases. This opens room for novel tools and techniques that may benefit from the study of API deprecation evolution.

II. Background

APIs are defined as interfaces used by client software components to communicate with a software provider entity [1]. Most industry level programming languages, such as JavaScript, rely on third-party library APIs to obtain reusable functionalities and tackle a wide variety of common problems.

JavaScript is a versatile programming language that conforms to the ECMAScript specification.[2] It has been primarily

[1]https://octoverse.github.com
[2]https://www.ecma-international.org

designed and known as a language for rendering dynamic content on the client-side of Web applications. More recently, JavaScript has also been used as a server-side language through the use of the Node.js environment.[3] Software reuse has become a key factor for a cost and time efficient software development project [8]. This scenario has led to the emergence of software repositories, such as npm, that provide a centralized and simplified management and distribution of software packages. Such packages hosted in npm provide reusable functionality to client applications via APIs.

Unlike other popular programming languages, such as Java and C#, JavaScript provides no native deprecation mechanisms. In addition, there are no recommendations from ECMA International or TC39 on how to properly deprecate JavaScript code. Thus, numerous deprecation mechanisms are used, alone or combined, in JavaScript packages, such as project documentation, console message, JSDoc annotation, and code comment [7]. Despite the growth on the usage of JavaScript external libraries and APIs, little is known how such API deprecation mechanisms are introduced and maintained over time. Additionally, to our best knowledge, there are no detailed studies aiming at analyzing API deprecation evolution in the JavaScript ecosystem.

III. Study Design

A. Selecting Candidate JavaScript Packages and Versions

To answer the research questions presented in the introduction, we analyzed popular JavaScript packages. We then investigated how deprecation evolved and when deprecated APIs are usually introduced or removed. We start by selecting the top-50 JavaScript packages sorted by the number of dependents on *Libraries.io*, a popular discovery service that indexes data from packages managers.[4] They track package releases, project's code, dependencies, and other useful information about open-source projects from a wide variety of programming languages. Thus, *Libraries.io* is a good data source for highly dependent JavaScript packages. We used the RESTful API[5] provided by *Libraries.io* to list the top-50 JavaScript packages with the most dependents count.

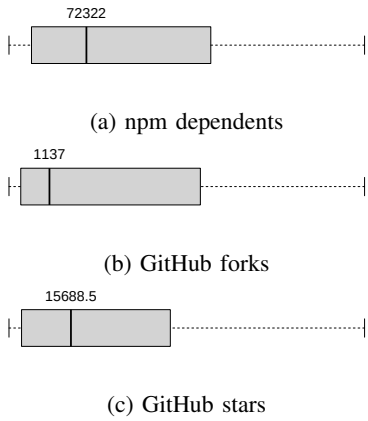
Figure 1 presents some statistics of the selected projects, retrieved in July, 2021, from `Libraries.io`. It presents box plots with the number of npm dependents (Figure 1a), and GitHub forks (Figure 1b), and GitHub stars (Figure 1c). The selected projects are highly popular in terms of dependents (median of 72.3K), forks (median of 1.1K), and stars (median of 15.7K). The top-3 libraries with most dependent have 280K, 235K, and 172K clients, respectively.

For each project, we downloaded all versions, considering their latest patch releases. We also removed unstable versions (such as alpha and beta) to avoid possible inconsistencies in the number of deprecated APIs between releases.

[3]https://nodejs.org/en/

[4]https://libraries.io/platforms

[5]https://libraries.io/apiproject-search



(a) npm dependents

(b) GitHub forks

(c) GitHub stars

Fig. 1: Overview of the selected JavaScript packages

B. Searching for Deprecation Occurrences

We downloaded the source code of all versions from the 50 selected packages, up to their latest stable version on July 2021. Next, we searched for all occurrences of the substring *deprecat* on JavaScript files to find possible deprecation candidates in all package versions, as recommended by prior literature [7]. We also tried the keyword *obsolete*, but only an insignificant number of occurrences were found, and none of them indicated deprecation. Thus, we focused on *deprecat* occurrences only. Furthermore, we only considered main source code files, excluding test, minified, and non-JS files (e.g., CSS and HTML) to avoid noisy data.

Next, we rely on Flow,[6] a well-known JavaScript code parsing library maintained by Facebook, to parse each file containing deprecation candidates, and to generate their corresponding abstract syntax trees (ASTs). For each AST, we visit its nodes to find deprecation occurrences of four types: deprecation utility, code comment, JSDoc annotation, and console message (as defined by prior research on JavaScript API deprecation [7]). For each package, we compute how many deprecated APIs each version has.

Furthermore, to answer RQ2, we assess whether the number of deprecated APIs tends to increase or decrease between releases and in which type of release it occurred (major or minor according to Semantic Versioning).[7]

C. Identifying Deprecation Trends

Finally, using the historical number of deprecated APIs for each project, we use the Mann-Kendall Trend Test (MK Test) to verify if there is an upward or downward trend on the deprecation occurrences. The MK test evaluates whether a set of historical values tend to increase or decrease over time. It is based on a non-parametric form of monotonic trend regression analysis. To perform the trend test, we relied on the `pymannkendall` package [9]. It exposes a function that receives a vector of values and outputs a named tuple

[6]https://flow.org

[7]https://semver.org

containing the trend (increasing, decreasing, or no trend), the p-value of the significance test, the Mann-Kendal's score S, and the Kendall τ value. The null hypothesis of no trend is rejected when S and τ are significantly different from zero.

IV. Results

A. *RQ1*. How do deprecated APIs evolve in JavaScript packages?

Table I presents the deprecation trend results of the analyzed packages. The first column indicates the trend results for each package. The second and third columns describe the package names and the number of analyzed releases. In the last column, we display line plots representing the evolution in the number of deprecated APIs. We found 18 packages (36%), out of 50, with no deprecation occurrences (shown as *No Deprec.*). From 32 packages with deprecation occurrences, 22 (69%) packages present statistically significant trends (p-value $>$ 0.05): 19 (59.4%) suggest an upward trend, while 3 (9.4%) packages indicate a downward trend. Finally, 10 (31.2%) packages present no statistically significant trend.

In addition to the overall number of deprecated over time, we also investigated how specific deprecation mechanisms evolve. In particular, we consider the deprecation mechanisms code comment, JSDoc annotation, console message, and deprecation occurrence. As summarized in Table II, the first column presents the deprecation mechanisms, while the second, third, and fourth columns present the trend results. The total of packages in the last row is greater than the sum of packages with deprecated APIs (32) because the same package might adopt more than one mechanism.

Considering the deprecation *code comment*, we found 29 packages with deprecation occurrences. From those, 17 packages (58.7%) present an increasing trend in the usage of this mechanism, while 5 (17.2%) show a decreasing trend. Regarding the deprecation *JSDoc annotation*, we detected 8 packages with deprecation occurrences: 5 (62.5%) with an upward and 1 (12.5%) with a downward trend. For *console message*, there were 17 packages with deprecation occurrences: 4 (23.5%) with an upward and 5 (29.4%) with a downward trend. Finally, we detected 20 packages with *deprecation utility*: 14 (70%) with an upward trend.

> ***RQ1 summary***: Close to 60% of the analyzed packages present an increase in the number of deprecated APIs, while only 9.4% show decreasing trends. In particular, 70% of the packages with *deprecation utility* present upward trends. On the other hand, the deprecation mechanisms with higher downward trends is *console message* (29.4% of the packages).

B. *RQ2*. To what extent are deprecated APIs introduced and removed in release types?

Table III presents the number of major and minor releases launched among the analyzed packages, and the number of

TABLE I: Mann-Kendall Trend Test results of deprecation occurrences on analyzed packages

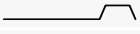
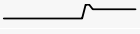
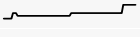
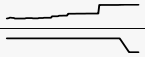
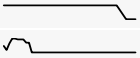
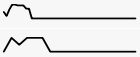
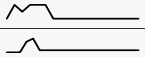
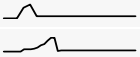
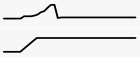
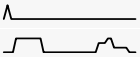
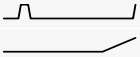
Trend	Package Name	# Releases	Evolution Plot
Upward	babel-loader	19	
	chai	31	
	commander	44	
	eslint	146	
	eslint-config-prettier	55	
	eslint-plugin-import	52	
	eslint-plugin-react	75	
	express	48	
	gulp	24	
	lint-staged	38	
	mocha	66	
	moment	38	
	prettier	36	
	react-dom	27	
	request	87	
	rollup	154	
	sinon	53	
	vue	18	
	webpack	132	
Downward	@babel/preset-env	15	
	lodash	48	
	webpack-cli	18	
No Trend	axios	21	
	babel-core	41	
	babel-preset-env	9	
	core-js	36	
	fs-extra	44	
	html-webpack-plugin	55	
	prop-types	5	
	react	38	
	vue-template-compiler	8	
	webpack-dev-server	45	
No Deprec.	@babel/cli	13	
	@babel/core	15	
	babel-cli	19	
	babel-eslint	18	
	babel-preset-es2015	13	
	babel-preset-react	10	
	chalk	15	
	coveralls	16	
	cross-env	11	
	css-loader	43	
	eslint-plugin-jsx-a11y	25	
	eslint-plugin-prettier	14	
	file-loader	21	
	husky	36	
	nyc	58	
	rimraf	10	
	sass-loader	29	
	style-loader	26	

TABLE II: Mann-Kendall Trend Test results of deprecation occurrences on analyzed, considering individual deprecation mechanisms

Deprecation Mechanism	Upward Trends	Downward Trend	No Trend
Code Comment	17 (58.7%)	5 (17.2%)	7 (24.1%)
JSDoc Annotation	5 (62.5%)	1 (12.5%)	2 (25.0%)
Console Messages	4 (23.5%)	5 (29.4%)	8 (47.1%)
Deprecation Utility	14 (70.0%)	0 (0.0%)	6 (30.0%)
Total	40	11	23

increased and decreased deprecated APIs. In total, we identified 127 major releases. In those releases, packages increased their number of deprecated APIs 57 times and decreased in 19 cases. Thus, we have an increase ratio of 0.45 and a decrease ratio of 0.15 deprecated API by major release. When we look at minor releases, we observe 1,357 launches. In those minor releases, they increased their number of deprecated APIs 1,246 times and decreased 718 times. Hence, we have an increase ratio of 0.92, and a decrease ratio of 0.53 deprecated API by a minor release. These results reveal that different from what the JavaScript community recommends [7], popular JavaScript packages usually add and remove deprecated APIs on minor releases instead of removing on major releases.

TABLE III: List of selected projects and number of versions

Release Type	Number of Releases	Number of Increased Deprecated APIs	Number of Decreased Deprecated APIs
Major	127	57 (0.45/release)	19 (0.15/release)
Minor	1,357	1246 (0.92/release)	718 (0.53/release)

RQ2 summary: Popular JavaScript packages usually add and remove deprecated APIs on minor releases instead of removing them on major releases. It is worth noticing that removing deprecated APIs in minor versions might cause developers to experience breaking changes on their projects since minor releases are not supposed to be breaking, according to Semantic Versioning.

V. Discussion and Implications

Overall, from our analysis of 1,918 releases of 50 JavaScript packages, results suggest that 60% of packages show rising trends in the number of deprecated APIs. This implies that as libraries evolve, and the number of APIs increase, more APIs might need to be deprecated. However, this result might also indicate that deprecated APIs are not consistently removed as new ones are added. Retaining old deprecated APIs might degrade code quality, and require more effort to maintain them. *For that reason, we suggest that JavaScript package maintainers should plan removal dates for deprecated APIs, and retain less deprecated code.*

When we look at deprecation approaches, we observe that several mechanisms are adopted to deprecate APIs. Additionally, such deprecation mechanisms are not consistently added and removed over time. For instance, the usage of deprecation utilities shows more upward trends (70%) when compared to other mechanisms. Although different deprecation mechanisms provide different functionalities, the lack of consistency makes it more complicated to maintain deprecated APIs. Additionally, different from what the JavaScript community recommends [7], our investigation suggests that popular JavaScript packages usually add and remove deprecated APIs on minor releases instead of removing on major releases. Removing deprecated APIs in minor versions might cause developers to experience breaking changes on their projects since minor releases are not supposed to be breaking. *Therefore, we argue that there is a strong need to create official deprecation rules and approaches for JavaScript APIs and make sure maintainers are consistent while implementing them.*

Lastly, JavaScript IDEs are currently not able to detect nor manage all mechanisms used for deprecation. Thus, package developers need to put a great effort into maintaining deprecated APIs. Having such management overhead, maintainers might overlook such APIs during package evolution. *This way, JavaScript developers could benefit productively from novel automated tools for helping maintainers manage deprecation (e.g., detecting retained deprecated APIs).*

VI. Threats to Validity

The study presented in this paper has some limitations that could potentially threaten our results, as we explain next. First, we focused the historical analysis on 50 popular JavaScript open-source packages, according to `Libraries.io`. Those packages might not be representative. To mitigate this threat, we selected widely adopted packages, with a large number of dependent clients.

Second, to identify deprecation occurrences, we only searched for matches of *deprecat*. We tried to find other occurrences by using the keyword *obsolete*, but only an insignificant number of occurrences were found, and none of them indicated deprecation. Thus, we focused on *deprecat* occurrences. Although being deliberate, this choice might have caused us to miss cases in which other terms are used.

Finally, the deprecation search script was implemented upon Flow, a well-known JavaScript code parsing library maintained by Facebook, and, thus, the risk of errors is reduced, but exists. Additionally, we have manually inspected its output (each sample with a confidence level of 95% and a confidence interval of 5%). We find a precision of 98% for deprecation utility, 81% for code comment, 100% for JSDoc comment, and 100% for console message. Thus, we considered that such results provided an acceptable level of confidence for this study.

VII. Related Work

Sawant et al. [6], [10], [11] conducted several studies to investigate Java API deprecation practices. The authors assessed the impacts, the needs, the reasons, and the patterns of API deprecation. They observed that the Java deprecation mechanism does not address all developers needs when it comes

to deprecation [10]. Brito et al. [12] investigated the use of deprecation messages in Java and C#. These studies describe that 66.7% and 77.8% of Java and C# API, respectively, are deprecated with deprecation messages and that this rate does not evolve. Li Li et al. [13] performed an exploratory study on Android API deprecation and identified that the Android framework is regularly cleaned-up from deprecated API, and their maintainers ensure that deprecated API is commented to provide replacement messages. Many other researchers study how API evolves, measure breaking changes, and analyze their impact on client systems [14] [15]. These studies mainly aimed at analyzing deprecation in Java and C#, but none focus on the JavaScript ecosystem.

Wang et al. [16] presented an explanatory study on Python library API deprecation and observed that API deprecation in Python is declared via three mechanisms: Decorator, Hard-coded warning, and comments. However, deprecation is poorly handled by library contributors and the usage of deprecated APIs is rarely changed. In our study, we observe that, besides not being standardized as well, deprecation APIs in JavaScript is not consistently maintained over time.

In the JavaScript ecosystem, Nascimento et al. [7] proposed a survey and a mining study to investigate deprecation mechanisms in the JavaScript ecosystem. The study reveals several deprecation mechanisms that might be used alone or combined, such as project documentation, console message, JSDoc annotation, code comment, and deprecation utility. Many other researchers study how API evolves, measure breaking changes, and analyze their impact on client systems [14], [15]. However, none of those studies cover API deprecation evolution in JavaScript. Thus, we complement the assessment of API deprecation in JavaScript with focus on evolution.

VIII. Conclusion and Further Steps

In this paper, we presented a historical analysis of API deprecation in popular JavaScript open-source packages. This work can help the software engineering community better understand how JavaScript deprecated APIs are maintained over time. By analyzing 1,918 releases of 50 popular JavaScript packages, we found that most packages (close to 60%) show rising trends in the number of deprecated APIs, while only 9.4% of the analyzed packages indicate a downward trend. As open-source packages evolve, these results suggest that deprecated APIs are not consistently removed as new ones are added.

When we look at specific deprecation mechanisms, we observe that the usage of deprecation utilities is increasing in most packages that adopt this mechanism. This mechanism is also the only one that shows no decreasing trend in any package. Additionally, we observe that the deprecation mechanism with the higher downtrends is console message. Finally, we note that most deprecation occurrences are both added and removed on minor releases.

We foresee extensions of this research as follows:

- Future work could aim at interviewing library maintainers to understand their rationale behind the adoption of deprecation mechanisms. Additionally, it could investigate why deprecated APIs seem to be retained rather than removed as projects evolve, and also why they are removed in minor releases rather than in major ones.
- Further steps could also include a tool to automatically identify deprecation, suggest deprecation messages, and alert developers about deprecated APIs.
- Lastly, further investigation of other characteristics of API deprecation in JavaScript, such as replacement messages and their structure, external documentation, and API evolution, could be conducted.

Acknowledgments

This research was partially supported by Brazilian funding agencies: CNPq, CAPES, and FAPEMIG.

References

[1] T. Tourwé and T. Mens, "Automated support for framework-based software," in *International Conference on Software Maintenance*. IEEE, 2003, pp. 148–157.

[2] S. Moser and O. Nierstrasz, "The effect of object-oriented frameworks on developer productivity," *Computer*, vol. 29, no. 9, pp. 45–51, 1996.

[3] W. Granli, J. Burchell, I. Hammouda, and E. Knauss, "The driving forces of API evolution," in *International Workshop on Principles of Software Evolution*, 2015, pp. 28–37.

[4] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: cost negotiation and community values in three software ecosystems," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.

[5] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem," in *International Symposium on the Foundations of Software Engineering*, 2012.

[6] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2158–2197, 2018.

[7] R. Nascimento, E. Figueiredo, and A. Hora, "JavaScript API Deprecation Landscape: A Survey and Mining Study," *IEEE Software*, pp. 1–15, 2022.

[8] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal API usage patterns," in *International Conference on Automated Software Engineering*. IEEE, 2011, pp. 456–459.

[9] M. M. Hussain and I. Mahmud, "pyMannKendall: a python package for non parametric Mann Kendall family of trend tests," *Journal of Open Source Software*, vol. 4, no. 39, p. 1556, 2019.

[10] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *International Conference on Software Engineering*. IEEE, 2018, pp. 561–571.

[11] A. A. Sawant, R. Robbes, and A. Bacchelli, "To react, or not to react: Patterns of reaction to API deprecation," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3824–3870, 2019.

[12] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "On the use of replacement messages in API deprecation: An empirical study," *Journal of Systems and Software*, vol. 137, pp. 306–321, 2018.

[13] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android APIs," in *International Conference on Mining Software Repositories*, 2018, pp. 254–264.

[14] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in APIs," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1458–1492, 2020.

[15] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2017, pp. 138–147.

[16] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library APIs are (not) handled," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 233–244.