# JavaScript API Deprecation Landscape: A Survey and Mining Study

Romulo Nascimento, Eduardo Figueiredo, Andre Hora

Deptartment of Computer Science, UFMG, Brazil
{romulonascimento, figueiredo, andrehora}@dcc.ufmg.br

**Abstract**

Software reuse is a key factor for a cost and time-efficient software development project. In JavaScript, this has led to the emergence of software repositories that provide reusable packages and libraries. As new versions of packages are released, their APIs might be deprecated. Consequently, client applications need to migrate to the newer API versions to benefit from new features and bug fixes. Unlike other popular programming languages, JavaScript provides no native deprecation mechanisms. To assess how developers deprecate JavaScript APIs, this paper reports the results of a survey with 109 JavaScript developers and a mining study on 320 popular JavaScript projects. Results suggest that there is no standard solution to deprecate JavaScript APIs. Overall, we find several solutions, including console message, project documentation, JSDoc annotation, code comment, and prefixed element. Furthermore, developers may use multiple deprecation solutions in the same project or even in the same file.

***Keywords***— API deprecation, JavaScript, software library

## 1 Introduction

JavaScript is a versatile programming language that has become extremely popular over the last years. According to the Stack Overflow Developer Survey,[1] JavaScript is the most commonly used programming language for the eighth consecutive year. GitHub Octoverse analysis[2] also shows
5 JavaScript as the most popular language in terms of repository contributors since 2014. Similar to other languages, software reuse has become a key factor for a cost and time efficient software

[1]https://insights.stackoverflow.com/survey/2020
[2]https://octoverse.github.com

development project in JavaScript [12]. This scenario has led to the emergence of software repositories that provide centralized management and distribution of software packages. npm is the most popular public repository of JavaScript packages. In 2019, it reached one million hosted packages,
10 making it the largest software repository to date [11]. Prior analysis suggests that 99% of JavaScript developers rely on npm to ease project dependency management.[3] This also points out the massive growth in npm usage that started about 5 years ago.

As new software versions are released, APIs may be renamed, updated, or removed, so developers can rely on deprecation mechanisms to warn their clients [3, 6, 8]. Consequently, client
15 applications should migrate to the latest stable or secure versions of their dependencies to benefit from novel features and bug-fixes. Other popular programming languages, such as Java and C#, provide native support mechanisms and tools to help developers explicitly deprecate APIs. Indeed, there has been much recent research to investigate deprecation practices on those languages [3, 4, 8, 10]. However, despite the growth in the usage of JavaScript libraries and APIs, little is
20 known about deprecation in JavaScript. Different from Java and C#, JavaScript provides no native deprecation mechanisms. Besides, to the best of our knowledge, there are no detailed studies regarding API deprecation in the JavaScript ecosystem.

To better understand the JavaScript API deprecation landscape, we propose a survey with developers and a mining study in JavaScript repositories. We reveal and discuss developers' solutions
25 to deal with deprecation in JavaScript by employing both quantitative and qualitative analyses. Our results are publicly available at: https://doi.org/10.5281/zenodo.4708397.

# 2 Survey with JavaScript Developers on API Deprecation

## 2.1 Survey Design

In practice, we find little academic literature on JavaScript API deprecation. Thus, to get initial
30 information on how developers deprecate JavaScript APIs, we rely on our preliminary study [5] as well as blogs, forums, and Q&A websites. This initial research revealed common approaches and discussions that helped us creating the survey questions. We targeted two perspectives: the API consumer (developers who use deprecated APIs) and the API provider (developers who maintain libraries and might deprecate APIs). As a result, we formulated five closed-ended questions to
35 capture both perspectives, as summarized in Table 1. Additionally, we added an open-ended question to encourage the respondents to add any thoughts, experiences, or suggestions they might have regarding API deprecation in JavaScript.

Next, we look for relevant candidates to answer the survey. This way, we search for and randomly select developers with contributions to JavaScript projects in GitHub. We then filter out

[3] https://www.businesswire.com/news/home/20190423005225/en/npm-Inc.-Survey-What's-New-in-JavaScript-2019

Table 1: Summary of survey questions to JavaScript developers

| # | Question Statements | Type | Options | Reasoning |
|---|---|---|---|---|
| | **API Consumer Perspective** | | | |
| 1 | How often do you see deprecated APIs or a deprecation message in JavaScript? | Closed-ended | Never<br>Occasionally<br>Sometimes<br>Often<br>Always | Understand if they have ever seen deprecated APIs and how often they perceive it.<br>If Never, it skips Questions 2 and 3. |
| 2 | What types of deprecation have you seen before? | Closed-ended | Options are taken from our first mining study [5] plus an "Other" option | Understand what known or unknown deprecation mechanisms they consume |
| 3 | How fast do you fix deprecation problems on projects you are working on? | Closed-ended | As soon as possible<br>Only when I have time<br>Only if it is necessary<br>Do not usually fix | Learn how developers react to deprecation and whether they consider it a critical issue |
| | **API Provider Perspective** | | | |
| 4 | How often do you deprecate the APIs (such as functions, classes, properties,etc) you implemented in JavaScript? | Closed-ended | Never<br>Occasionally<br>Sometimes<br>Often<br>Always | Understand if they have ever deprecated APIs and how often they deprecate.<br>If Never, it skips Question 5. |
| 5 | How do you deprecate the APIs you implemented in JavaScript? | Closed-ended | Options are taken from our first mining study [5] plus an "Other" option | Understand what known or unknown deprecation strategies they use |
| 6 | Is there any thoughts, suggestions or experiences you would like to share about deprecation mechanisms and practices in JavaScript? | Open-ended | | Collect additional experiences or suggestions they might have on API deprecation |

40 developers with less than 50 commits in the last year, to only keep the ones who have been actively working in JavaScript recently. We also remove developers with more than 100 followers, as very popular developers could be less likely to respond to surveys. We end up with a list of 14,480 emails. We send 100 survey emails daily to developers until we reach at least 100 responses.

We first conducted a pilot survey with nine developers to validate the proposed questions and 45 evaluate if they are adequate and clear. That helped us minimize the risk of sending the survey to a large number of developers with ambiguous questions. After the pilot survey, we added a "Documentation" option to Questions 2 and 5. This option was not initially considered and appeared on four answers. The pilot answers were not considered in the final results.

## 2.2 Survey Results

50 We sent survey invitation emails to 1,389 developers and had 109 responses, which corresponds to an 8% response rate. Table 2 summarizes the answers to the five open-ended questions. The first three questions of the survey address the developers as API consumers, while the last two questions as API providers.

Table 2: Survey closed-ended questions and answers

| | API Consumer Perspective | # | % | | API Provider Perspective | # | % |
|---|---|---|---|---|---|---|---|
| **Q1** | **How often do you see deprecated APIs in JavaScript?** | | | **Q4** | **How often do you deprecate APIs you implement in JavaScript?** | | |
| | Never | 8 | 7.3% | | Never | 22 | 20.2% |
| | Occasionally | 41 | 37.6% | | Occasionally | 58 | 53.2% |
| | Sometimes | 34 | 31.2% | | Sometimes | 18 | 16.5% |
| | Often | 21 | 19.3% | | Often | 8 | 7.3% |
| | Always | 5 | 4.6% | | Always | 3 | 2.8% |
| **Q2** | **What types of deprecation have you seen before?** | | | **Q5** | **How do you deprecate APIs you implement in JavaScript?** | | |
| | Console message | 89 | 88.1% | | Documentation | 44 | 50.6% |
| | Documentation | 75 | 74.3% | | JSDoc | 42 | 48.3% |
| | JSDoc | 49 | 48.5% | | Console message | 40 | 46.0% |
| | Code comment | 25 | 24.8% | | Code comment | 37 | 42.5% |
| | Prefixed element | 23 | 22.8% | | Utility | 14 | 16.1% |
| | Other | 6 | 6.0% | | Prefixed element | 14 | 16.1% |
| | | | | | Deprecation list | 3 | 3.4% |
| | | | | | Semantic Versioning | 3 | 3.4% |
| | | | | | Other | 2 | 2.2% |
| **Q3** | **How fast do you fix deprecation problems?** | | | | | | |
| | As soon as possible | 25 | 24.8% | | | | |
| | Only when I have time | 34 | 33.7% | | | | |
| | Only if it is necessary | 37 | 36.6% | | | | |
| | Do not usually fix | 5 | 5.0% | | | | |

### 2.2.1 The API Consumer Perspective

**Q1. How often developers see deprecated APIs in JavaScript projects**. Only 4.6% of the respondents always see deprecated APIs during development activities, while 19.3% notice them often. In contrast, 31.2% of the respondents see deprecated API only sometimes and 37.6% occasionally. Lastly, 7.3% have never seen deprecated APIs in JavaScript. Notice that a large proportion of the developers (76.1%) see deprecated APIs less frequently, either sometimes, occasionally, or never.

**Q2. What deprecation mechanisms developers have seen before**. In our survey, the most common deprecation solutions mentioned by the developers were console messages (88.1%) and project documentation (74.3%). Also, 48.5% of the developers have seen deprecated elements annotated with the JSDoc annotation `@deprecated` and 24.8% have noticed them in single code comments. Moreover, 22.8% have seen APIs prefixed with any sort of deprecation prefixes, such as `deprecated__`. Finally, six developers revealed other solutions: four developers mentioned they have seen deprecation console messages specifically during the package/library installation, one respondent indicated deprecation error messages at runtime, and one added the usage of the custom Deprecated<T> type in TypeScript. Overall, this result suggests that, from the consumer perspective, deprecation messages delivered via console messages and project documentation are more likely to be perceived by developers.

**Q3. How fast developers fix deprecation problems**. 24.8% of the respondents state they fix deprecation issues as soon as possible, 33.7% fix only when they have time, and 36.6% fix only if necessary. Lastly, a minority of the developers (5%) do not usually fix deprecation issues.

### 2.2.2 The API Provider Perspective

75 **Q4. How often developers deprecate APIs in JavaScript projects**. Only 2.8% of the developers always deprecate APIs, while 7.3% often deprecate. While 16.5% deprecate sometimes, over a half (53.2%) deprecate APIs occasionally. Lastly, 20.2% have never deprecated APIs.

**Q5. What deprecation mechanisms developers use to deprecate API**. The majority of the API providers (50.6%) use the project documentation to inform about deprecated APIs. Next,
80 we find three categories with similar ratios: 48.3% annotate deprecate elements with the JSDoc annotation `@deprecated`; 46% use console messages to warn about deprecated API; and 42.5% add code comments next to API to indicate deprecation. Moreover, 16.1% of the developers use utilities to aid API deprecation and 16.1% prefix API elements to indicate they have been deprecated. Lastly, 3.4% maintain, somewhere within the project, a list/object of deprecated elements, while 3.4% state
85 they remove deprecated APIs on major releases, following the Semantic Versioning specification.[4] One developer added they rely on the npm-deprecate[5] npm CLI command to deprecate the whole package and another one indicated the usage of the custom `Deprecated<T>` TypeScript type.

## 2.3 Developers' Further Insights

In the last question, we encouraged participants to share thoughts on JavaScript deprecation. In
90 this part, we received 20 answers and used thematic analysis to analyze the responses.

The current state of practice of deprecation in which maintainers tend to retain deprecated APIs in favor of clients was criticized by three developers. In this case, one developer noted: "*the current practice/implementation of deprecation in JS breeds a culture of complacency on top of old and dangerous systems*", causing an endless backward compatibility effort that degrades code health.
95 Moreover, one developer indicated that, with the fast and ever-evolving JavaScript ecosystem, "*deprecation of API is time-consuming and laborious*", but it is nevertheless beneficial.

Developers also emphasized that deprecation communication does not necessarily need to happen at the API level, since major releases are expected to bring breaking changes. For example, one developer stated: "*when we roll out a new major version of a library, the changes are always*
100 *breaking*". In those cases, any required upgrade should preferably be communicated on release notes.

As package maintainers, two developers suggested that console messages are the most efficient way to communicate deprecation. However, consumer developers argued they would like to be able to suppress deprecation messages, even if temporarily.

[4]https://semver.org
[5]https://docs.npmjs.com/cli/deprecate

Additionally, three developers emphasized the importance of clear and constant communication about deprecated APIs. For example, one developer stated: "*for any consumed APIs, as much deprecation communication as possible is preferred*", either with an internal team member or with client systems. Finally, four developers suggested a cohesive deprecation strategy as a appropriate way to approach deprecation on a project: "*your versioning strategy is the way you inform your consumers what is the scope of a change via the version number*".

To conclude, we present four deprecation best practices suggested by a developer: (1) plan a deprecation strategy and make clients aware of it; (2) use Semantic Versioning to communicate breaking changes; (3) inform clients about upcoming changes via project documentation and console logs, preferably with a message containing a target date and release and a link to a migration guide; and (4) if a deprecated API need to be kept, either add a `unsafe__` or similar prefix or provide them through a opt-in flag, such as `-legacy` or `-insecure`.

# 3 Mining API Deprecation in JavaScript

## 3.1 Mining Study Design

Next, we conducted a mining study to analyze deprecation in popular JavaScript libraries and understand how package maintainers deprecate APIs. We selected the top-320 most depended upon projects in the npm registry to build our library dataset. The npm most depended upon packages are highly popular projects in the JavaScript community since it is currently the most adopted public collection of JavaScript packages. For example, the top-3 libraries have 131K, 67K and 65K dependents, respectively.

We downloaded the source code of the 320 selected projects, considering their latest stable versions in March 2021. We then searched for all occurrences of the substring *deprecat* in JavaScript files to find possible deprecation candidates. While navigating in the project files, we only considered the main source code files, excluding test, minified, and non-JavaScript files (e.g., CSS and HTML). Next, we use the Flow[6] tool, a well-known JavaScript code parsing library maintained by Facebook, to parse each file containing deprecation candidates and generate their corresponding abstract syntax trees (ASTs). By analyzing ASTs, we can programmatically detect code comments and their content, function declarations and their identifier names, expression calls and their components, and many other components of code structure and language syntax.

Based on the most frequent mechanisms indicated by developers in both the survey and our preliminary study [5], we focused on categorizing deprecation occurrences of four types: deprecation utility, code comment, JSDoc annotation, and console message (see Listing 1). We rely on the ASTs

[6]https://flow.org

```javascript
const deprecate = require("depd")("my-math-module");

function addDigits(x, y) {
    deprecate('addDigits is deprecated. It will be deleted on version 3.0.')
    return x + y;
}
```

(a) Deprecation utility: any sort of code specially written to aid deprecation at any complexity level

```javascript
function addDigits(x, y) {
    // Function deprecated since version 2.3. It will be deleted in version 3.0.
    return x + y;
}
```

(b) Code comment: use of code comments, excluding occurrences of JSDoc

```javascript
/**
 * @deprecated since version 2.3. It will be deleted on version 3.0.
 */
function addDigits(x, y) {
    return x + y;
}
```

(c) JSDoc: use of the @deprecation JSDoc annotation

```javascript
function addDigits(x, y) {
    console.warn("addDigits is deprecated. It will be deleted on version 3.0.");
    return x + y;
}
```

(d) Console message: use of the JavaScript engine native console API

Listing 1: Examples of JavaScript deprecation approaches

obtained from the analyzed JavaScript files to automatically find occurrences of each deprecation type, as follows:

1. **Deprecation utility:** function declaration or call in which the function identifier name matches the substring *deprecat*, as presented in Listing 1a.

140 2. **Code comment:** any type of code comment that matches the substring *deprecat* excluding occurrences of JSDoc annotations, as shown in Listing 1b.

3. **JSDoc annotation:** the exact usage of the JSDoc `@deprecated` annotation inside a comment, as presented in Listing 1c.

4. **Console message:** calls to console functions (such as warn, log, and error) in which the
145 message argument is a string literal that matches the substring *deprecat*, as shown in Listing 1d.

We manually evaluated samples of each category to measure the precision of our tool to correctly identify API deprecation (each sample size ensured a confidence level of 95% and a confidence interval of 5%). Each deprecation case was evaluated by the first author of the paper and validated by the co-authors. In case of conflict, all authors discussed until an agreement was reached. We find a precision of 98% for deprecation utility, 81% for code comment, 100% for JSDoc comment, and 100% for console message. This way, our tool can be used with a good level of confidence.

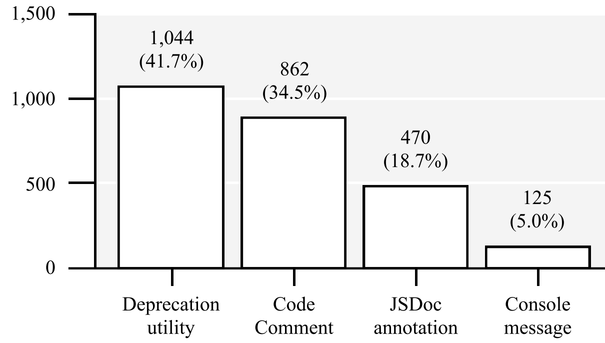
## 3.2 Mining Study Results

We observed deprecation occurrences on 122 out of the 320 analyzed projects. Considering those 122 projects, we found 2,501 deprecation occurrences in 681 out of 35,318 files.

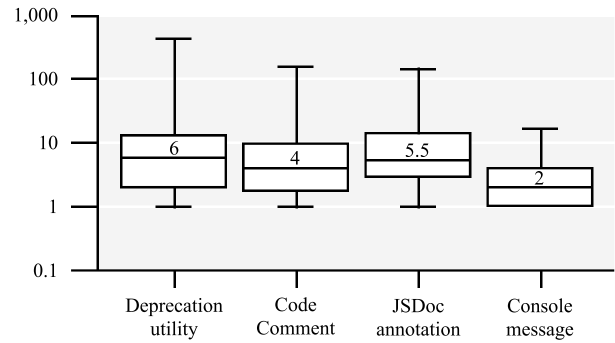
Figure 1a presents the deprecation occurrences by category. The most frequent deprecation mechanism is *deprecation utility* (41.7% of the cases), which represents any sort of code function specially written to support deprecation. This category is followed by *code comment* (34.5%) and *JSDoc annotation* (18.8%). Lastly, the direct usage of *console message* is the least common (4.7%). It is important to note that, although *deprecation utility* accounts for almost half of deprecation occurrences present in subject projects, fewer projects adopt this strategy when compared to *code comments*. Project *@alifd/next* comprises 41.4% (432) of the 1,044 deprecation utility occurrences. We believe that this explains why fewer developers indicated the usage of deprecation utilities as opposed to other strategies in our survey results described in Table 2.

Figure 1b presents the distribution of the deprecation per project. The median values range from 2 (*console message*) to 6 (*deprecation utility*). Indeed, projects that adopt *deprecation utility* tend to have more deprecation occurrences and may have specific deprecation needs that are not satisfied by other simpler mechanisms.

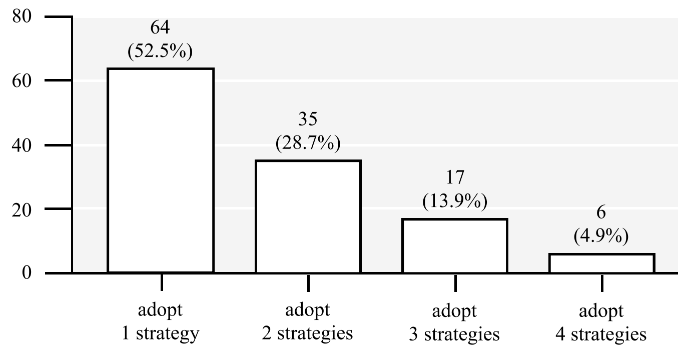
Next, Figure 1c summarizes the combination of deprecation strategies per project. Over half of the analyzed projects (52.5%) adopt only one deprecation mechanism, while over 1/4 (28.7%) combine two deprecation mechanisms. As we increase the number of combined mechanisms, the number of packages decreases. For example, we only found the occurrence for the four studied mechanisms in 6 (4.9%) packages. Figure 1d presents a detailed view of this data: a Venn diagram showing the intersection of deprecation mechanisms. This detailed view is presented in two levels of granularity: package and file level. In both package and file levels, the most adopted single strategy is *code comment* (38 packages and 216 files). The most common combination is *deprecation utility* and *code comment*, which is present in 16 packages and 39 files. Although they are not adopted by many packages, *deprecation utility* and *JSDoc annotation* are highly used at the file level, with 174 and 123 occurrences, respectively. This suggests that projects that adopt those two strategies tend to use those mechanisms very frequently. Also, similarly to what occurs at the package level, most files implement a single deprecation mechanism. The two most frequent combinations at file level



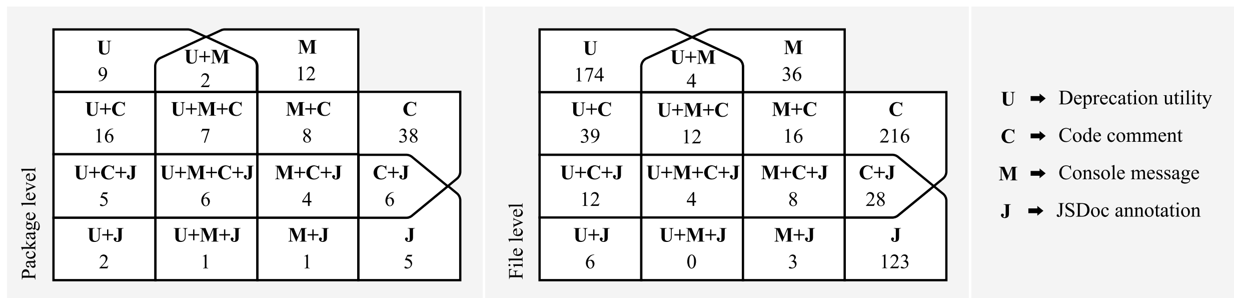
(a) Deprecation occurrences by category



(b) Occurrences distribution by category



(c) Number of packages by the number of deprecation strategies adopted



(d) Number of packages and files that adopt mechanism combinations

Figure 1: API deprecation mechanism occurrences in JavaScript packages

are *deprecation utility* and *code comment* (39 files) and *code comment* and *JSDoc annotation* (28 files).

In summary, we find no standard solution to deprecate JavaScript APIs in the wild. Moreover, 185 we observe that the four studied deprecation strategies (deprecation utility, code comment, JSDoc annotation, and console message) are used both standalone or combined at project and file levels.

# 4 Implications

Our empirical findings reveal possible implications related to the deprecation of JavaScript APIs. This provides opportunities for future research to better support the JavaScript community in han- 190 dling deprecation.

**Guidelines for deprecating JavaScript APIs.** The lack of a clear and consistent strategy to deprecate JavaScript APIs brings challenges for maintainers, consumers, and researchers. As presented in our survey and mining studies (Table 2 and Figure 1), we find that package maintainers adopt different strategies and use a diverse set of ad-hoc solutions. Thus, API maintainers need to 195 set clear deprecation strategies for their APIs and follow them consistently.

**Official rules to better support deprecation in JavaScript.** Library maintainers follow different approaches, standalone or combined, to deprecate their APIs. However, those are not necessarily effective approaches. Console messages, for instance, might not be noticed or annoy developers. Developers would thus benefit from an official and effective deprecation mechanism, with well-defined 200 rules, to support library maintainers and the JavaScript community. For example, native support solutions, similar to the deprecation mechanisms in Java and C#, could help JavaScript developers.

**Novel tools to automatically detect the usage of deprecated APIs.** Popular JavaScript IDEs are currently not able to detect all mechanisms used to deprecate APIs. Indeed, this may happen due to the inconsistency of the adopted deprecation strategies. This way, JavaScript developers 205 could benefit productively from novel tools to detect the most common deprecation mechanisms, so that they become more aware of the API deprecation in their projects.

# 5 Threats to Validity

First, we focused the analysis on 320 JavaScript open-source projects hosted in npm, the most popular JavaScript package manager. Despite these observations, our findings cannot be generalized 210 to other systems implemented in other languages or closed-source packages. Additionally, we analyzed packages with a large number of dependent clients, as we expect them to be examples of well maintained projects and representative case studies of open-source packages with many dependent clients. However, their maintainers may not represent the whole population of JavaScript developers.

Furthermore, survey findings cannot be directly generalized since the participants might not be representative of the general population of JavaScript developers outside GitHub. Future replications should address these issues.

Second, the survey questions might have been unclear or ambiguous. To minimize this threat, we conducted a pilot survey to collect feedback and improve the survey. Furthermore, developers might have provide unreliable answers. For example, provided deprecation strategies or reactions might deviate from reality. To minimize this threat, we send a short and focused survey, informing that it had research purposes only.

Third, to identify deprecation occurrences, we only searched for matches of *deprecat*. We tried to find other occurrences by using the keyword *obsolete*, but only 18 out of 35,318 files 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. Furthermore, since we mine all JavaScript files, the deprecation strategies we analyze might also relate to internal APIs that are not visible to consumers. Future replications that select external APIs only should address this issue. Moreover, the JavaScript tool for the mining study was implemented upon Flow, a well-known JavaScript code parsing library maintained by Facebook, thus, the risk of errors is reduced. Additionally, we have manually inspected its output (each sample with a confidence level of 95% and a confidence interval of 5%) and we found a high precision. Thus, our tool can be used with a good level of confidence.

Finally, the deprecation mechanism options used in the survey might bias respondents by limiting their ideas of what deprecation looks like. However, we also provided an "other" option to encourage respondents to add other solutions. Also, the experimental observations and analyses were manually conducted by the authors of the paper, therefore, the results be subjective. To mitigate this threat, we adopted thematic analysis.

# 6 Related Work

## 6.1 Previous Studies on API Deprecation

Several studies investigate Java API deprecation [2, 3, 4, 7, 8, 9, 10], with a focus on the impact, the needs, the reasons, and the patterns. For example, Sawant et al. observed that Java deprecation mechanisms do not address all developer needs [7]. By mining other data sources, such as source code, issue tracker data, and commit history, the authors have found 12 reasons that trigger developers to deprecate API [8]. The same research group also verified that most API client applications do not react to deprecation [10]. Brito et al. [2, 3] investigated the use of deprecation messages in Java and C#, finding that 66.7% and 77.8% of Java and C# APIs are deprecated with deprecation messages and that this rate does not evolve. Li Li et al. [4] performed an exploratory study on Android API deprecation. The authors found that the Android framework is regularly cleaned up

from deprecated API and their maintainers ensure that deprecated APIs are commented to provide
250 replacement messages. Other ecosystems are also analyzed, for example, Robbes et al. [6] studied deprecation in the context of Smalltalk. Wang et al. [13] conducted an explanatory study in the Python ecosystem, finding that API deprecation is poorly handled by library contributors and the usage of deprecated APIs is rarely changed. Yasmin et al. [15] proposed a framework to investigate RESTful API deprecation and revealed that 87.3% of breaking changes were not deprecated in
255 previous versions. Recently, we have published a preliminary study in the context of JavaScript deprecation [5], which was the basis of the survey and mining study described in this study. Many other researchers study how API evolves, measure breaking changes, and analyze their impact on client systems [1, 14].

## 6.2 Comparison with Other Languages

260 The analysis of deprecation mechanisms have targeted different programming languages, such as Java [2, 3, 4, 7, 8, 9, 10], C# [3], Python [13], and Pharo/Smalltalk [6]. We first note that different from JavaScript, Java, C# and Pharo/Smalltalk have built-in deprecation mechanisms. Java developers can annotate an element with `@Deprecation` to mark an obsolete API, triggering compilers to warn about deprecated API usage. Additionally, Javadoc also provides an `@deprecated` tag that helps
265 generate API documentation. Other studies have shown that code comments and documentation notes are also common deprecation strategies in the Java ecosystem [7].

Pharo/Smalltalk implements a deprecation strategy in which developers can provide a message output at run-time as a warning. As in Java, a high number of projects do not update to newer versions after an API deprecation had been introduced [6]. C# has the `Obsolete` attribute to
270 flag deprecated API elements [3]. While there is no deprecation built-in mechanism in Python, developers can rely on the *Deprecated*[7] project, which provides the `@deprecated` decorator and logs deprecation warnings. However, in Python, library maintainers tend to use different decorators or adopt ad-hoc local solutions; alternative strategies, such as hard-coded warnings and comments, are also common [13]. As in other languages, deprecated APIs are not usually addressed by developers
275 during the evolution of Python projects [13].

In JavaScript, the JSDoc annotation `@deprecated`, for instance, is equivalent to the one defined by the Javadoc specification. Other mechanisms, such as warning messages and code comments, are also similar to code conventions found in other programming languages.

# 7 Conclusion

280 We summarize our major findings as follows:

[7]https://pypi.org/project/Deprecated

1. We find no standard approach to deprecate JavaScript APIs.
2. Overall, the most commonly adopted deprecation mechanisms are console message, project documentation, JSDoc annotation, and code comment.
3. Developers usually learn about deprecated JavaScript APIs via console message and project documentation.
4. Most JavaScript developers only address deprecation issues if necessary or if time permits.
5. Distinct deprecation strategies may be used in the same project or file.

As future work, we plan to investigate other characteristics of API deprecation, such as replacement messages and the adoption of external documentation. Furthermore, our survey study brought to our attention the practice of introducing breaking changes communicated by other means, such as Semantic Versioning, in preference to API deprecation. We hypothesize that the fast-moving JavaScript community might prefer such approaches in favor of package publication speed. However, we wonder to what extent JavaScript developers are aware of Semantic Versioning to update project dependencies. That also remains a future work plan.

# References


[1] 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*, 25:1458–1492, 2020.

[2] G. Brito, A. Hora, M. T. Valente, and R. Robbes. Do Developers Deprecate APIs with Replacement Messages? A Large-scale Analysis on Java Systems. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 360–369. 2016.

[3] G. Brito, A. Hora, M. T. Valente, and R. Robbes. On the Use of Replacement Messages in API Deprecation: An Empirical Study. In *Journal of Systems and Software, vol. 137*, pages 306–231. 2018.

[4] 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 (MSR)*, pages 254–264. 2018.

[5] R. Nascimento, E. Figueiredo, A. Hora, and A. Brito. JavaScript API Deprecation in the Wild: A First Assessment. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 567–571. 2020.

[6] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to API deprecation?: the case of a Smalltalk ecosystem. In *International Symposium on Foundations of Software Engineering (FSE)*. 2012.

[7] 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 (ICSE)*, pages 561–571. 2018.

[8] A. A. Sawant, G. Huang, G. Vilen, S. Stojkovski, and A. Bacchelli. Why are Features Deprecated? An Investigation into the Motivation Behind Deprecation. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 13–24. 2018.

[9] A. A. Sawant, R. Robbes, and A. Bacchelli. On the Reaction to Deprecation of 25,357 Clients of 4+1 Popular Java APIs. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 400–410. 2016.

[10] A. A. Sawant, R. Robbes, and A. Bacchelli. To react, or not to react: Patterns of reaction to API deprecation. In *Empirical Software Engineering, vol. 24*, pages 3824–3870. 2019.

[11] L. Tal and S. Maple. npm passes the 1 millionth package milestone! What can we learn? In *https://snyk.io/blog/npm-passes-the-1-millionth-package-milestone-what-can-we-learn. Last access: Sep, 2020.*

[12] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal API usage patterns. In *International Conference on Automated Software Engineering (ASE)*, pages 456–459. 2011.

[13] J. Wang, L. Li, K. Liu, and H. Cai. Exploring How Deprecated Python Library APIs Are (Not) Handled. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020.

[14] 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 (SANER)*, pages 138–147. 2017.

[15] J. Yasmin, Y. Tian, and J. Yang. A First Look at the Deprecation of RESTful APIs: An Empirical Study. In *International Conference on Software Maintenance and Evolution (ICSME)*. 2020.