Towards Better Comprehension of Breaking Changes in the NPM Ecosystem

DEZHEN KONG*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JIAKUN LIU*, School of Information Systems, Singapore Management University, Singapore
LINGFENG BAO†‡, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
DAVID LO, School of Information Systems, Singapore Management University, Singapore

Code evolution is prevalent in software ecosystems, which can provide many benefits, such as new features, bug fixes, security patches, etc., while still introducing breaking changes that make downstream projects fail to work. Breaking changes cause a lot of effort to both downstream and upstream developers: downstream developers need to adapt to breaking changes and upstream developers are responsible for identifying and documenting them. In the NPM ecosystem, characterized by frequent code changes and a high tolerance for making breaking changes, the effort is larger.

For better comprehension of breaking changes in the NPM ecosystem and to enhance breaking change detection tools, we conduct a large-scale empirical study to investigate breaking changes in the NPM ecosystem. We construct a dataset of explicitly documented breaking changes from 381 popular NPM projects. We find that 95.4% of the detected breaking changes can be covered by developers' documentation, and 19% of the breaking changes cannot be detected by regression testing. Then in the process of investigating source code of our collected breaking changes, we yield a taxonomy of JavaScript and TypeScript-specific syntactic breaking changes and a taxonomy of major types of behavioral breaking changes. Additionally, we investigate the reasons why developers make breaking changes in NPM and find three major reasons, i.e., to reduce code redundancy, to improve identifier names, and to improve API design, and each category contains several sub-items.

We provide actionable implications for future research, e.g., automatic naming and renaming techniques should be applied in JavaScript projects to improve identifier names, future research can try to detect more types of behavioral breaking changes. By presenting the implications, we also discuss the weakness of automatic renaming and breaking change detection approaches, such as the lack of support for public identifiers and various types of breaking changes.

CCS Concepts: • **Software and its engineering** → *Software libraries and repositories*; **Software evolution**.

Additional Key Words and Phrases: Breaking Change, NPM, JavaScript, Code Evolution

1 INTRODUCTION

The evolution of code is prevalent in software ecosystems [34, 35, 46, 53]. Developers of upstream software libraries make code changes to incorporate new features, bug fixes, security patches, component refactorings, and

*Both authors contributed equally to the paper.
†Corresponding author.
‡Also with Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute.

Authors' addresses: Dezhen Kong, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, timkong@zju.edu.cn; Jiakun Liu, School of Information Systems, Singapore Management University, Singapore, Singapore, jkliu@smu.edu.sg; Lingfeng Bao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, lingfengbao@zju.edu.cn; David Lo, School of Information Systems, Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg.

ACM 1557-7392/2024/11-ART
https://doi.org/10.1145/3702991

extra-functional improvements [51, 53]. However, code evolution may break the contract previously established with its downstream by introducing *breaking changes* (BCs) in its public APIs, making client applications fail to work [55]. For example, renaming frequently used methods or classes can make client projects that depend on the code fail to compile. Therefore, downstream developers will be required to make efforts to adapt to such breaking changes.

Considering the risks and the effort brought by breaking changes to downstream developers, many works have been utilized to help developers analyze and measure the impacts of breaking changes [4, 40, 41]. However, they relied on test suites in downstream projects, and either detected JavaScript breaking changes by directly running test suites [4], or via API models[1] generated from dynamic execution of test cases [40, 41]. This leads to the result that breaking changes observed in prior studies are located in well-known APIs that are popularly used in downstream projects, such as `Client.socket` in socket.io[2], `Request.prototype.onResponse` in request[3], and `each` in async. However, breaking changes are reported to upstream projects because they already manifested themselves in downstream test cases or already caused bugs. For example, in version 3.0.0 of a famous JavaScript library lodash, the behavior of function `mixin` was changed and not explicitly marked by upstream developers, and a downstream developer reported this issue by self-constructed test code.[4] Hence, completely relying on test case execution is not sufficient for understanding breaking changes.

We observe that in the NPM ecosystem, a number of projects write commit messages complying with Conventional Commits [2]. According to Conventional Commits, BREAKING CHANGE tokens are documented in commit messages to indicate that the corresponding commits contain breaking changes. This is because the core developers of upstream libraries need to ensure the stability of the API and protect the reputation of the upstream projects. When a new code change is submitted for review, they carefully inspect it for the presence of breaking changes, as well as other bugs or defects. By leveraging the documented breaking changes provided by developers, there is a potential opportunity to notify downstream users about these breaking changes.

However, developers experience increased pressure within the NPM ecosystem, primarily driven by frequent code changes and a high tolerance for breaking changes [13, 14]. Another possible factor is that different developers may have different perceptions of breaking changes and some of them may ignore the breaking changes, leading to unaware breaking changes that cause unforeseen bugs in downstream projects. If we could characterize the large number of documented breaking changes at source code level, it would assist upstream developers in gaining a deeper understanding of the types of commits that could potentially introduce breaking changes. Furthermore, such categorization would provide valuable insights for researchers to develop tools aimed at detecting breaking changes in the future.

To bridge this gap, we conduct an empirical study to better comprehend documented breaking changes. We first select popular projects with over 50 GitHub stars and associated links to an existing GitHub repository, resulting in a total of 35,786 repositories. We then randomly select 381 repositories (95% confidence level and 5% margin of error) for further investigation. We then clone their associated repositories. After this, we identify BC-related commits admitted by developers by searching for BREAKING CHANGE tokens in commit messages. For the 5,242 identified BC commits, we manually select 1,519 commits that 1) do not contain too long commit messages, 2) are related to JavaScript production code and 3) are associated with documentation that can explain the reasons behind the commit. These selected BC commits are distributed across 131 distinct projects and the projects can be grouped into six categories according to functionalities and usages, i.e., utilities, frontend projects, development-related tools, database-related tools, plugins and Web API related tools (detailed in Section 3), which makes the BC-related commits in our projects representative. To extract breaking changes from these BC

[1]Intuitively, an API model defines the type restrictions of an API [40].
[2]https://github.com/socketio/socket.io/commit/b73d9be
[3]https://github.com/request/request/commit/d05b6ba
[4]https://github.com/lodash/lodash/issues/880

commits, we build breaking change type taxonomy by learning from a previous API detection tool for Java [16] and adding new JavaScript and TypeScript-specific features. Our taxonomy includes Remove, Rename, Change Signature, Change Behavior types, etc., which is detailed in Section 2.2. Through several thematic analyses, we obtain JavaScript and TypeScript specific code features (RQ2 and RQ3) and a taxonomy of reasons behind breaking changes. We answer the following research questions in this study:

RQ1: To what extent do *detected breaking changes* and *documented breaking changes* overlap? In our work, we collect breaking changes from developers' intention, i.e., the explicitly documented breaking changes. We first check whether documented breaking changes actually break test code. Therefore, we do regression testing on our collected BC-related commits, similar to previous works [40–42, 56]. We find 95.4% of detected breaking changes are documented and 81% of the documented breaking changes can be detected by regression testing. The result shows that most detected breaking changes are well documented, while a proportion of documented breaking changes cannot be detected by regression testing. Since the documented breaking changes cover most of the detected breaking changes, it is reasonable to extract breaking changes from documentation (especially commit messages, issues, and pull requests).

RQ2: What syntactic breaking changes in the NPM ecosystem are specific to JavaScript and TypeScript? We investigate whether there are some breaking changes related to JavaScript-specific language features. Despite the triviality of most syntactic breaking changes (such as moving classes to another package, removing a field of a class), we find some JavaScript-specific BCs are notable, including two removing operations, i.e., 1) removing default export, 2) removing export of an element in one place, and five types of JavaScript and TypeScript specific signature changes, e.g., parameter changes in configuration objects, switch between callback and Promise, etc.

RQ3: How do developers make behavioral breaking changes? In this RQ, we investigate how developers perform breaking changes at source code level. Since syntactic breaking changes can be detected through syntax analysis and code refactoring detection, we mainly focus on behavioral breaking changes (a.k.a, semantic breaking changes in some works) that change the internal program logic. We find four major types of Change Behavior breaking changes, i.e., 1) changing the specification return values, 2) changing process for some option values, 3) changing default or initial value of variables and 4) changing error handling method.

RQ4: Why do developers make breaking changes in NPM ecosystem? In this RQ, we revisit the rationale of breaking changes in the NPM ecosystem and find three main factors (each with some sub-factors) that motivate developers to make breaking changes, i.e., 1) to reduce code redundancy, 2) to improve identifier names, 3) to improve API design, and divide each reason into some sub-items, which extends the previous works [13–15].

Based on our empirical findings, we provide actionable implications for future research, including 1) automatic renaming deserves much concern in JavaScript projects since poor identifier naming is a main contributor to technical debt, 2) automatic tools for ensuring code consistency are necessary, 3) future research should strive to detect more types of behavioral breaking changes. By presenting these implications, we also discuss the weakness of automatic renaming and breaking change detection approaches, such as lacking support for public identifiers and various types of breaking changes.

The contributions of our paper are two-fold:

(1) We build a carefully constructed breaking change dataset extracted from a wide range of JavaScript and TypeScript projects in the NPM ecosystem.
(2) We empirically identify how developers perform breaking changes at source code level, yielding notable findings like JavaScript and TypeScript specific syntactic breaking changes and typical actions of behavioral breaking changes. We also investigate why developers perform breaking changes in the NPM ecosystem.

The remainder of this paper is organized as follows. Section 2 introduces preliminary knowledge and motivating examples. Section 3 presents the methodology of our study. Section 4 details our empirical findings. Section 5

provides our implications for future research from this study. Section 6 discusses the threats to validity. Section 7 reviews the related work and Section 8 concludes our work.
Data availability. We provide the replication package of our research at https://doi.org/10.5281/zenodo.13927690.

2 BACKGROUND

In this section, we describe some preliminary knowledge on breaking changes.

2.1 Preliminary of Breaking Changes

Prior to our work, a number of studies have uncovered breaking changes from many aspects, such as the occurrence of breaking changes, the impacts of breaking changes on downstream projects, and motivations for making breaking changes. The research works [22, 47, 54] have demonstrated the widespread occurrence of breaking changes in the NPM ecosystem and their effects on client applications. Venturini et al. found that in their sampled packages, 11.7% of all client packages and 13.9% of their releases are impacted by breaking changes, and notably, 44% of the breaking changes are introduced in minor or patch releases (according to Semantic Versioning [11], developers should not introduce breaking changes in non-major releases) [54]. Bogart et al. found that coarse-grained motivations for making breaking changes include requirements and context changes, bugs and new features, rippling effects from upstream changes, and technical debt from postponed changes [13, 14].

However, despite much knowledge of BC, to the best of our knowledge, none of the studies precisely defined BC. Researchers collected breaking changes in various forms in previous studies. For example, in the evaluation of several breaking change detection tools [40, 56], the authors collected breaking changes by running downstream test cases: if a test case of a downstream project could not pass with a newer provider (after the code change), then the code change was identified as a BC. However, on the one hand, not all providers are dependent on many client applications. If a code change has no triggering test case in client applications, we cannot determine whether it is actually incompatible. On the other hand, client code may not follow the specification of providers and incorrectly access APIs, e.g., passing an improper value to a function, which will result in undefined behavior [41], including test failure, but it does not reflect a BC. In another breaking change detection tool APIDiff [16], Brito et al. utilized the code refactoring detection tool RefDiff [49] to identify syntax-related breaking changes (e.g., changing method signature or renaming class). However, as they pointed out, the detected breaking changes are just breaking change candidates (BCC), since some syntax changes are applied to internal methods and classes, which are not intended for public use. To this end, they also asked developers to check whether the detected BCCs are actually breaking changes [15].

In our study, we adopt Brito et al.'s criteria [15], since we also investigate the breaking changes from developers' perspective, i.e., how and why they perform breaking changes. Specifically, a breaking change should 1) be confirmed by developers, 2) be categorized into pre-defined BC types. To achieve the first criterion, we identify breaking changes by searching explicit BC declarations in the documentation (usually commit messages). To achieve the second criterion, we define several breaking change types for JavaScript and TypeScript on the basis of Brito et al.'s Java BC types [16] and JavaScript development experience. We detail the BC types in Section 2.2 and describe how to extract breaking changes in Section 3.3.

2.2 Types of Breaking Changes in NPM Projects

Existing BC detection tools for Java, typically APIDiff [16], support many syntactic and object-oriented programming (OOP) related BC types, such as *remove classes* and *push down fields*. Since OOP features are also supported in JavaScript (since ECMAScript 2015 [5], `class` can be directly used while legacy code must leverage the prototype mechanism [7]), we learn from the breaking change types for the Java language and adapt them to JavaScript and TypeScript (a superset of JavaScript, in practice many source code files are written in TypeScript and will be

Table 1. Breaking Change Types Adopted from Previous Works

Type	Supported Elements
Rename	module, class, interface, enum, type, method, field, constant, variable
Remove	module, class, interface, enum, type, method, field, constant, variable
Move	module, class, interface, enum, type, method, field, constant, variable
Inline	method
Push Down	method, field
Change Signature	method, field
Change Behavior	the content of method, field, constant and variable

compiled into JavaScript files) by retaining available BC actions in JavaScript and adding more breaking change types which are not considered previously. We then construct a taxonomy of JavaScript and TypeScript breaking changes (shown in Section 1), where *interface*, *enum* and *type* are only available in TypeScript. We explain each type as follows:

(1) **Rename** refers to identifier changes of public code elements. Rename BCs can be applied to many code elements, such as exported classes, interfaces, and enums, as well as methods and fields in exported classes and interfaces.
(2) **Remove** refers to removal of public code elements. Remove BC can be applied to many code elements. After a Remove BC, the affected public element is not accessible by downstream applications, e.g., an exported class is completely removed, or no longer marked as exported.
(3) **Move** can be applied to all APIs supported by Rename BCs. For example, a class is moved to another package since its functionality is more related to that package.
(4) **Inline** is a refactoring operation in object-oriented programming: removing a public method and copying its body into an existing method. Simple methods are often inlined to make code more straightforward. In fact, Inline can be regarded as a special case of Remove, since the inlined public method is removed from public access. By contrast, normal "Remove" actions do not copy the content of the removed method to another existing method. However, since inline is a very common refactoring operation in object-oriented languages, we still regard it as a separate breaking change type, like previous works [16, 49].
(5) **Push Down** is also a refactoring operation in object-oriented programming: moving the method in the parent class into the child class since this method is only used in one child class in reality. Push down can be regarded as a special case of Remove BC.
(6) **Change Signature** refers to the modification of method signatures except for directly changing method names. Adding or removing modifiers (such as "static", "private") and changing the parameters are typical refactoring actions of this type.
(7) **Change Behavior** (also called *semantic breaking changes* in some works [40, 56]) refers to those operations changing the internal behavior (e.g., programmatic logic in methods, and content of global constants that can be used by some methods) rather than syntactic elements, such as changing class names and adding required parameters. In other words, after a behavioral breaking change is performed, the affected classes (or methods, interfaces, etc.) can be invoked as the same way as before.

In our work, we focus on breaking changes *in JavaScript source code*, hence we do not take breaking changes in non-source code files into consideration, typically CSS, Markdown, HTML, `package.json` files and configuration files related to TypeScript, Webpack and ESLint, etc.

3 METHODOLOGY

3.1 Data Collection

Due to the vast number of JavaScript projects in NPM, our study focuses on selecting the most popular projects to ensure that the collected breaking changes are representative. To do so, we utilize the Libraries.io open source repository and dependency metadata provided by Reid et al. [48] to retrieve the most popular JavaScript and TypeScript projects since Libraries.io is commonly used in prior studies [23, 24, 28]. We first select popular projects having more than 50 GitHub stars and associated with an existent GitHub repository, which yields 35,786 repositories. We then randomly select 381 repositories (95% confidence level and 5% margin of error) for further investigation.

Since we consider the breaking changes that are explicitly confirmed by developers, we try to extract breaking changes from developers' documentation, typically commit messages, *changelogs*, issues and pull request text on GitHub. We notice that Conventional Commits [2] provides a standard for writing readable commit messages. According to the standards, if a commit message contains a `BREAKING CHANGE` section, the commit is identified as a breaking change. For example, the text below shows a typical commit message indicating a breaking change:

```
refactor: compiler -> runtimeCompiler
BREAKING CHANGE: compiler option has been renamed to runtimeCompiler
```

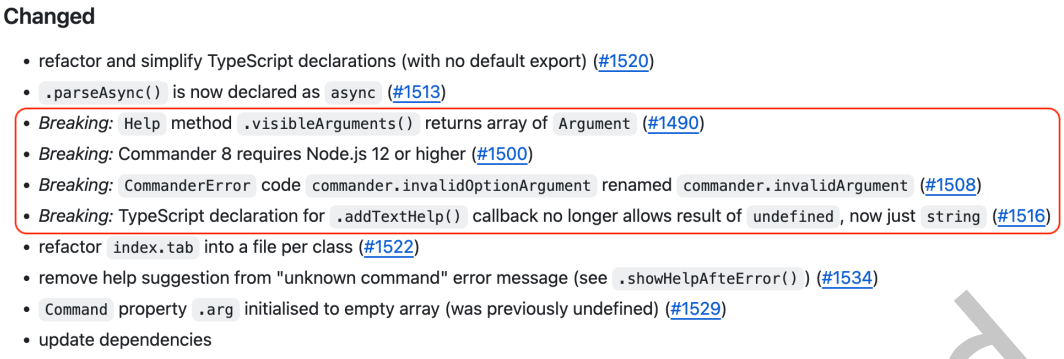
We first check whether Conventional Commits are widely used, analyzing the commits from the 381 projects we used. We find that 360 out of 381 repositories contain commits that conform to Conventional Commits and 198 projects have over 80% commits that follow Conventional Commits. We use the regular expression below to check Conventional Commit compliance:

```
(fix|feat|chore|build|ci|test|style|perf|refactor)(\([a-zA-Z0-9-\s]+\))?!?:
```

Here `fix`, `feat`, `chore`, etc., are the defined or recommended scope tokens in Conventional Commits specification. This indicates that *Conventional Commits* specification is widely used and we can use it as a simple way to extract plenty of breaking changes. Considering that many projects do not follow this standard, we also try our best to extract BC commits from *changelogs* of each project (if developers have declared), issues and pull requests on GitHub. A *changelog* summarizes the changes from the last version release, and may mention multiple breaking changes. By analyzing the *changelog*, we can link the mentioned breaking changes to corresponding commits. For example, the project tj/commander.js does not use Conventional Commits, then we consult the mentioned pull request IDs in `CHANGELOG.md` file[5] (shown in Figure 1, where four breaking changes are mentioned), and we also analyze the content of the pull requests to obtain the commits related to the breaking changes. In this way, we obtain 5,242 commits.

Then two of the authors manually inspect them and remove some of them that are not related to JavaScript production code (e.g., only modifying package.json, Markdown documentation, build scripts, test cases) or contain very long commit messages (over 10 lines) that are difficult to understand. For example, if a commit only states that "drop support for Node.js 14" and modifies the corresponding item in `package.json`, it will be discarded. Additionally, we do not include the commits that contain multiple breaking change declarations. Specifically, we remove 1,826 commits that contain over 10 line commit messages or multiple BREAKING CHANGE declarations and 692 commits that are not JavaScript code changes. Finally, we retain 2,724 candidate commits.

[5] https://github.com/tj/commander.js/blob/master/CHANGELOG.md



Changed

- refactor and simplify TypeScript declarations (with no default export) (#1520)
- `.parseAsync()` is now declared as `async` (#1513)
- *Breaking:* `Help` method `.visibleArguments()` returns array of `Argument` (#1490)
- *Breaking:* Commander 8 requires Node.js 12 or higher (#1500)
- *Breaking:* `CommanderError` code `commander.invalidOptionArgument` renamed `commander.invalidArgument` (#1508)
- *Breaking:* TypeScript declaration for `.addTextHelp()` callback no longer allows result of `undefined`, now just `string` (#1516)
- refactor `index.tab` into a file per class (#1522)
- remove help suggestion from "unknown command" error message (see `.showHelpAfteError()`) (#1534)
- `Command` property `.arg` initialised to empty array (was previously undefined) (#1529)
- update dependencies

Fig. 1. Screenshot of the Changelog of commander.js 8.0.0 (The four highlighted breaking changes can be linked to related issues and commits)

3.2 Regression Testing

We use regression testing to detect breaking changes in our selected projects (381 in total) to check:

(1) what percentage of detected breaking changes are well documented,
(2) what percentage of documented breaking changes can be detected by regression testing.

To check ①, due to the number of the total commits being too large (1,005,344), we first sample 16,371 commits for further analysis (99% confidence level and 1% margin of error). We then remove the commits in these projects that only contain non-JavaScript code changes (e.g., test code changes, Markdown and HTML documentation changes, dependency updates in `package.json`). Then for each commit c and its prior commit c' we do regression testing following the steps below:

(1) We run the test cases in c' directly (using `npm test` command). We discard the commits that cannot be configured (e.g., there are version conflicts during installation).
(2) We restore the production code from c. The production code refers to JavaScript and TypeScript source code files not located in `test` directory.
(3) We run the test cases (using `npm test` command).
(4) If the first step succeeds and the third step fails, then we say that c is a detected BC commit.
(5) If a detected BC commit is documented with BREAKING CHANGE, or can be found in *changelog*s, issues, or pull requests, we regard the commit as a documented BC commit.

We use the test cases written by developers rather than dependents since 1) we have found that 363 out of the 381 projects provide test suites, and developers' test cases can better express the intended usage of APIs [25], while many of the 381 projects do not have plenty of dependents, 2) some projects are not intended for pragmatic use, especially CLI tools in our projects such as npm/cli, therefore no clients access the APIs in these projects. To check ②, we also apply the steps described above to the 2,724 BC-related commits from Section 3.1.

3.3 Breaking Change Selection and Analysis

3.3.1 Breaking Change Selection and Categorization. We select feasible breaking changes for further research from the 2,724 candidate BC commits and categorize them into types described in Section 2.2. Since we need to understand *what* a commit changed and *why* developers made it, we require the commit message and other documentation of a commit (including issues, pull requests and *changelogs*) to contain some explanation of *why*

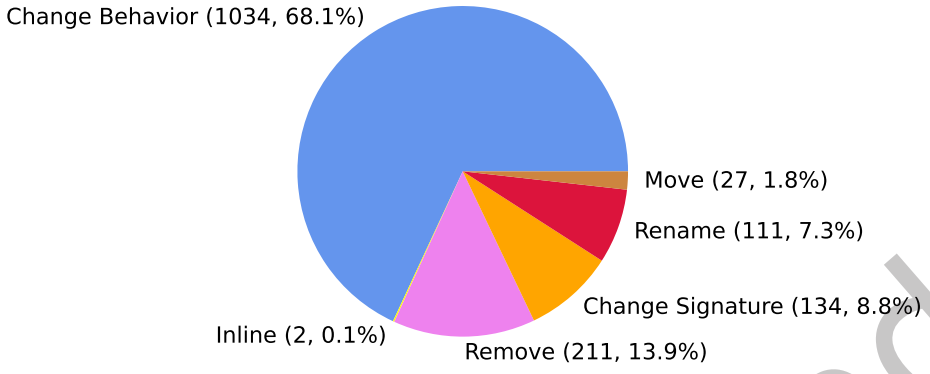


Fig. 2. Categories of Our Investigated Breaking Changes

developers made that commit. We here reuse Tian et al.'s taxonomy of *why* information in commit messages [52]. Typical *why* information includes:

(1) Issue description: about the linked issue, weakness of current code implications, etc.
(2) Requirements Illustration: about the usage need, out-of-date statements, etc.
(3) Objective description: objectives, such as fixing bugs, improving performance, refactoring code, etc.
(4) Necessity Implication: relation to prior commits, benefits of making such code changes, etc.

For example, considering the commit cebd670a of angular/angular[6], the developers just stated renaming the method `requestCheck` in `ChangeDetectorRef` to `markForCheck`. By inspecting the related issue (#3403), the developer said that "When I first saw the requestCheck() method on ChangeDetectorRef, I assumed it was how I manually run change detection. Others have assumed this as well", which is a description of "Necessity Implication".

If there is no *why* information covering the aspects mentioned above, we discard the commit. Two of the authors independently check whether the documentation contains reason-related information (Cohen's Kappa value is 0.80) and determine the type of each commit (using the taxonomy in Section 2.2, Cohen's Kappa value is 0.77). Then the authors hold meetings to solve the disagreements. In total we identify 1,519 commits that contain reason-related information.

3.3.2 Breaking Change Distribution. Figure 2 presents the percentages of different types of breaking changes. It can be obviously seen that Change Behavior BCs make up the largest proportion of all BC types (68.1%). By contrast, Move and Inline BCs make up the smallest percentage (1.8% and 0.1% respectively). In our further investigation, we mainly focus on Change Behavior BCs.

The projects that contain breaking changes in our study serve various types of functionalities, including command-line interface utilities, REST API SDKs and Web frontend frameworks. They can be grouped into six categories (shown in Table 2). For example, pnpm/pnpm[7] is a alternative CLI tool for official NPM implementation,

[6]https://github.com/angular/angular/commit/cebd670a
[7]https://github.com/pnpm/pnpm

Table 2. NPM Projects Containing Breaking changes Used in Our Study

Category	Description	Number
Frontend	Used in Web browser environment	45
Web API	Providing access to RESTful APIs	9
Database Tools	Providing database operations	4
Development Tools	Providing project management	25
Plugin	Served as plugins for other NPM projects	3
Utility	Other useful JavaScript libraries	45

Table 3. Proportion of Involved Projects of Each Breaking Change Category

BC Category	Frontend	Utility	Development	Database	Web API	Plugin	# All
Remove	10/45	12/45	8/25	4/4	4/9	0/3	38/131
Rename	10/45	10/45	3/25	3/4	3/9	0/3	29/131
Move	2/45	2/45	1/25	1/4	1/9	0/3	7/131
Inline	0/45	0/45	1/45	0/4	0/9	0/3	1/131
Change Signature	8/45	10/45	6/25	3/4	2/9	0/3	29/131
Change Behavior	44/45	41/45	25/25	4/4	9/9	3/3	130/131

renovatebot/renovate provides automatic project build and aws/aws-cdk[8] wraps the AWS cloud APIs, gajus/eslint-plugin-flowtype[9] is an ESLint plugin, and async/async[10] is a fundamental utility of asynchronous programming. Table 3 shows the number of involved projects for each type of breaking changes.

3.3.3 Breaking Change Labeling. For the 1,519 remaining commits, two of the authors conducted two thematic analyses to analyze ① source code features (for RQ2 and RQ3) and ② reasons behind the commit (for RQ4). The guideline recommended by Cruzes et al. [21] were used. The guideline for the two analyses is shown below. Two of the authors independently performed the steps above, and they held a series of meetings to solve the agreements.

(1) The authors read the related documentation, especially commit messages carefully to understand what the developers want to express.
(2) The authors read the documentation again and generate phrases as initial codes that describe the source code features, how developers make breaking changes and the reasons behind the commit.
(3) The authors aggregated the codes with similar meanings and generate a theme name to describe each cluster.
(4) The authors then reviewed all themes and try to merge the themes with similar semantics, or made the similar themes become sub-items of a new theme.

After obtaining all themes, the authors assigned possible themes to each breaking change. Note that for simplicity, we assigned each retained breaking change with one reason, which is consistent with the previous

[8]https://github.com/aws/aws-cdk
[9]https://github.com/gajus/eslint-plugin-flowtype
[10]https://github.com/async/async

works on motivations behind breaking changes [15]. In this process, the authors held several meetings to resolve the disagreements on the reason behind each commit since the reasons behind some BC commits are difficult to determine. For example, in the commit 2713380 of reactivex/rxjs[11], developers renamed `inspect` to `audit`, the reasons behind this rename operation was presented in Issue #1505[12] and #1387[13]. Developers mentioned two possible reasons, i.e., name collision (since `inspect` conflicts with `util.inspect` provided by Node.js[14]) and synonym replacement (since `audit` is synonymous to `inspect` and it is less confusing). By carefully reading and understanding the related issues, two of the authors reach an agreement and assign the reason "to avoid name conflict" to this breaking change, because if `inspect` were not used in Node.js native objects, the identifier `inspect` is still acceptable and does not need to be renamed. In the process of assigning the reason to the breaking changes, the authors have very few disagreements (the Cohen's Kappa value 0.94).

4 RESULTS

In this section, we present our empirical results for each research question.

4.1 RQ1: To What Extent do Detected Breaking Changes and Documented Breaking Changes Overlap?

From 16,371 commits used in regression testing in Section 3.2, we detect 173 BC commits (1.0%), 95.4% of them (165) are documented by developers. And for 2,724 BC commits, 2,206 (81%) can be detected by regression testing. The results illustrate that most detected breaking changes are well documented, while a proportion (19%) of documented breaking changes cannot be detected via regression testing since developers might forget to write test cases or current test cases are too general to cover the code modifications.

The undocumented breaking change commits are possibly due to ① the commit is made too long ago when Conventional Commits was not proposed, and ② the project does not comply to Conventional Commits. And the main reasons that regression testing cannot detect breaking changes is that the BC behavior can only be triggered by external conditions (e.g., network error), hence it is difficult to simulate such a situation. For example, in commit 61e7a81a of octokit/rest.js, the `followRedirects` option is no longer supported. This BC can only manifest itself when the HTTP server returns a status code between 301 and 307. However, this is not often seen, and developers have not written any test case to cover the process of "followRedirects" option. Developers might forget to write test cases and current test cases might be too general to cover the code modifications, hence automatic test case generation may be useful.

For the 19% of breaking changes that cannot be detected, since developers in popular projects have rich development experience, we believe those breaking changes are trustable and worth attention. Although some breaking changes currently do not affect many downstream projects, they should still be pointed out since in some cases downstream developers may write code that invokes the broken API, then the downstream developers can confirm the breaking changes quickly by looking up commit messages. For example, in commit 669592d of socketio/socket.io[15] contains a breaking change that removed `Socket#binary` method. It is not detected when we run regression testing. However, after several months, one developer asked for an alternative method of `Socket#binary`[16]. This indicates that although an API in a breaking change is not frequently used, and currently no test cases can trigger it, the API is possibly invoked at a certain time.

[11] https://github.com/reactivex/rxjs/commit/2713380
[12] https://github.com/reactivex/rxjs/issues/1505
[13] https://github.com/reactivex/rxjs/issues/1387
[14] See explanation in Issue 1387 of Rxjs.
[15] https://github.com/socketio/socket.io/commit/669592d
[16] https://github.com/socketio/socket.io/discussions/3826

4.2 RQ2: What Syntactic Breaking Changes in NPM Ecosystem Are Specific to JavaScript and TypeScript?

While most code-level actions in breaking changes are also available in other programming languages, such as changing parameter orders and removing classes, etc., we highlight the noticeable syntactic BC actions in our investigated projects.

4.2.1 JavaScript-specific remove operations. In ECMAScript 6, a module can have many exported items and a default export item. Therefore, developers can not only remove the code of a class, interface or enum, etc. like other OOP languages, but also just remove them from exported list and put them into internal source code files. We present the JavaScript-specific Remove operations in our collected breaking changes as follows:

Remove a default export (6 cases). ECMAScript 6 supports default export functionality[17]. For example, in commit 102e4b0 of nodkz/mongodb-memory-server, developers simply removed default export of the module util. Therefore, the code `import generateDbName from 'util'` will not work.

Remove an export position (17 cases). For example, in commit cfbfaac of reactivex/rxjs, the types in `rxjs/interfaces` modules are no longer accessible and users must use them by importing them from the main module. Therefore, users can import `UnaryFunction` with the following two methods, while after the code change, they can only import `UnaryFunction` using the second form.

```
1 import { UnaryFunction } from 'rxjs/interfaces';
2 import { UnaryFunction } from 'rxjs';
```

4.2.2 JavaScript-specific signature changes. Most signature changes in JavaScript are also available in other languages like Java, e.g., adding, removing, and reordering parameters. However, some JavaScript-specific signature changes should be focused on. We present the categories as follows:

About parameters in configuration objects (15 cases). In JavaScript, many properties can be packaged in one configuration object. Considering the *diff* shown below, the second parameter of the method `userinfo` is an object, and this function only extracts four properties in this object using destructuring assignment syntax [3]. After the code change, the `verb` property is renamed to `method`, since `GET` is an HTTP method. Correspondingly, the internal code related to this property is also changed.

```
  async userinfo(accessToken,
-   { verb='GET', via='header', tokenType, params } = {}
+   { method='GET', via='header', tokenType, params } = {}
  ) { /* ... */ }
```

About `this` parameter (6 cases). In the old implementation, the method accepts a callable object (typically a function) and `this`, and binds the function to `this`[18]. After the code change, the method no longer provide this parameter, the user should first bind the callable object to `this`. For example, in the code snippet shown below, `findIndex` only accepted `predicate` parameter.

```
   export function findIndex<T>(
     predicate: (value: T, index: number, source: Observable<T>) => boolean,
-     thisArg?: any
   ): OperatorFunction<T, number> {
-    return operate(createFind(predicate, thisArg, 'index'));
```

[17] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

[18] `bind` is a native function in JavaScript. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

```
+     return operate(createFind(predicate, undefined, 'index'));
  }
```

About sync and async (20 cases). In JavaScript, asynchronous functions can be implemented by using callback arguments and Promise objects. Legacy JavaScript asynchronous functions are designed with callback-style interface, while new implementations often use `async` directly. For example, in the commit 8c3cecae of automattic/mongoose, developers provided multiple signatures of an API before the code change, e.g., `createIndex` have the following function signature:

```
1 createIndexes(options: mongodb.CreateIndexesOptions, callback: CallbackWithoutResult):
      void;
2 createIndexes(callback: CallbackWithoutResult): void;
3 createIndexes(options?: mongodb.CreateIndexesOptions): Promise<void>;
```

After the code change in this commit, the signature 1 and 2 were removed. The code *diff* below shows this pattern:

```
-  SomeFunction(param, (returnValue, err) => {
-    // Some code about returnValue
-  }
+  let returnValue = await SomeFunction(param);
+  // Some code about returnValue
```

Another case is converting synchronous to asynchronous (or vice versa), e.g., simply adding `async` modifier to the function signature.

About undefined and null (8 cases). The keyword null represents an intentional empty value, while undefined occurs in accessing uninitialized variables or non-existent object properties, and should not be explicitly assigned to an object. The *diff* below shows this pattern:

```
-  public get(key: Key): Value | null {
-    return this.collection.get(key) || null;
+  public get(key: Key): Value | undefined {
+    return this.collection.get(key);
   }
```

However, in JavaScript projects without `d.ts` files, there are also signature related changes, e.g., a method no longer accepts a null value, or a null value might cause error after a code change. In this case, developers cannot represent the change in method signature, therefore, we classify the case as behavioral change.

Switch between "required" and "optional" of fields in method or interface declarations (3 cases). TypeScript script projects and part of JavaScript projects use `d.ts` files to declare interface and method signature. For example, if one parameter is changed to be required, then a "?" will be removed from the parameter type. The *diff* below (from commit 86074a6 of coinbase/rest-hooks) shows the signature change type:

```
   static url<T extends typeof SimpleResource>(
     this: T,
-    urlParams?: Partial<AbstractInstanceType<T>>,
+    urlParams: Partial<AbstractInstanceType<T>>,
   ): string {
-   if (urlParams) {
-     if (
-      Object.prototype.hasOwnProperty.call(urlParams, 'url') &&
-      // ...
```

4.3 RQ3: How do Developers Make Behavioral Breaking Change?

We identify four main types of behavioral changes in our collected JavaScript projects of NPM ecosystem. Each type is presented as follows:

Changing the specification of return values (79 cases). One typical case is that the type of returned objects are changed. For example, in commit b99f6d3 of automattic/mongoose, the method `MongooseArray.map()` returned a plain JavaScript array rather than a headless Mongoose array:

```
   map() {
-    const ret = super.map.apply(this, arguments);
-    ret[arraySchemaSymbol] = null;
-    ret[arrayPathSymbol] = null;
-    ret[arrayParentSymbol] = null;
+    const copy = [].concat(this);

-    return ret;
+    return Array.prototype.map.apply(copy, arguments);
   }
```

Changing the process of some options (231 cases). Suppose that one possible value of an option (can be in JSON configuration or parameters) is not supported, then the code that gets properties from the option will be changed, and the corresponding branch of a specific option value will be removed. For example, in commit ad4f1493 of project octokit/rest.js[19] (shown below), the judgment of `options.type` is changed, and `netrc` branch in a switch structure is also removed to completely remove support for `netrc` mechanism.

```
   if (!options.type ||
-   'basic|oauth|client|token|integration|netrc'.indexOf(options.type) === -1
+   'basic|oauth|client|token|integration'.indexOf(options.type) === -1
    ) {
       throw new Error("Invalid authentication type, must be'' +
-        'basic', 'integration', 'oauth', 'client' or 'netrc'")
+        'basic', 'integration', 'oauth', or 'client'")
   }
```

In CLI tool projects, developers often change the process of some options to remove or change the purpose of an option. For example, in commit f6fd0c3 of pnpm/pnpm, the option `--store-path` is not an alias of `--store` anymore. Therefore, developers remove the code of handling the option, shown as follows:

```
  // in function run (argv: string[])
    await new Promise((resolve, reject) => {
      setTimeout(() => {
-       if (opts.storePath && !opts.store) {
-         logger.warn('the store-path config is deprecated.')
-         opts.store = opts.storePath
-       }

       // `pnpm install ``"` is going to be just `pnpm install`
       const cliArgs = cliConf.argv.remain.slice(1).filter(Boolean)
       // more code ...
```

[19] https://github.com/octokit/rest.js/commit/ad4f1493

Changing the default behavior for unprovided values (203 cases). The arguments of a function can often affect the behavior. When the value of a parameter is not provided (i.e., being `undefined`), developers may design special program logic to deal with this case, or make the parametre still remain undefined. Sometimes the default program logic of dealing with such case can be changed. As an example, in commit 72bbda7f of npm/cli[20], the default value of local variable `depthToPrint` is set to zero while it is initially `undefined`. After this change, the code is as follows:

```
const { /* ... */, depth, /* ... */ } = npm.flatOptions;
const depthToPrint = all ? Infinity : (depth || 0);
// more code
```

The default values can be also in global configurable objects. For example, Renovate puts configurations that affect the behavior in lib/config/options/index.ts, and the modification to it will cause behavioral change.

Changing error handling process (42 cases). One case is changing change the content or format of the error objects. For example, in commit dd6306e3 of octokit/rest.js[21], the error message is parsed as a JSON object and the properties will be put into error object (the existing `message` property will be overwritten). Since developers usually parse `error.message`, this change will make the code like `JSON.parse(error.message)` not work. The code *diff* below shows this change:

```
  // in a Promise chain
  .then(data => { /* ... */ })
  .catch(error => {
     if (error instanceof HttpError) {
+      try {
+        Object.assign(error, JSON.parse(error.message))
+      } catch (_error) {
+        // ignore, see #684
+      }
       throw error
   }
```

Another case of this category is changing the criteria of detecting errors, e.g, some status values are no longer regarded as success. In the commit 201f189d in angular/angular[22], if the return code of an XHR (XML HTTP Request) is not 200, an error will occur, while before the code change, if the return codes between 200 and 300 (exclusive) were all regarded as success states. The breaking change in `xhr_backend.ts` is shown as follows:

```
   let response = new Response(responseOptions);
+  if (isSuccess(status)) {
     responseObserver.next(response);
     responseObserver.complete();
+    return;
+  }
+  responseObserver.error(response);
```

The code in `http_util.ts` that contains the new anonymous function `isSuccess`:

```
+  export const isSuccess =
+    (status: number): boolean => (status >= 200 && status < 300);
```

[20]https://github.com/npm/cli/commit/72bbda7f
[21]https://github.com/octokit/rest.js/commit/dd6306e3
[22]https://github.com/angular/angular/commit/201f189d

Table 4. Typical Rename Operations

Category	Explanation	Number
Add token	add non-trivial tokens, e.g., from description to ariaDescription	23
Change token order	change orders in the identifier, e.g., secretJsonValue to secretValueFromJson	2
Remove token	remove one token from the identifier, e.g., renderBoundElementIndex to boundElementIndex	15
Replace	completely replace the token with another token, e.g., signed to trusted	20
Replace token	replace one token in identifier, e.g., fromDockerHub to fromDockerRegistry	39
Trivial	add `is` or `get` prefix, e.g., from failed to isFailed	12

We also compare the behavioral breaking changes to those in the Maven ecosystem. Zhang et al. have studied the behavioral breaking change types in Maven projects [56]. They found that changing execution logic and changing calculation of the output make up over 60% of the total breaking changes collected by them. They also studied benign changes that will not cause incompatibilities, such as additional/changed/deleted conditions and branches (over 55% of the total benign changes), additional try-catch statements, and assignment revision. However, in our findings about source code patterns of JavaScript behavioral breaking changes, it is very common to change conditions in "if" statements to remove support for some option values or to change evaluation criteria of some results, although it seems benign. Hence, in JavaScript, the condition changes that involve option/configuration values deserve attention since they are more probable to be breaking changes.

4.4 RQ4: Why do Developers Make Breaking Changes in NPM Ecosystem?

In this section, we summarize six major motivations behind making breaking changes by carefully analysis of our collected breaking changes. We show the connection between BC types and motivations in Figure 3.

Reason 1: To reduce code redundancy. (469 cases) Some source code is no longer needed, hence developers remove them. There are two cases of this reason:

- To unify the approach of a functionality (21 cases). The functionality can be achieved by other methods, or by using other packages. Therefore, the current implementation can be removed. For example, in commit 1354171 of thi-ng/umbrella, developers removed function `writeFile`, and recommend using `writeFile` provided by another package @thi-ng/rstream-log-file since the third-party function can also achieve the same functionality.
- To deprecate and remove an outdated functionality (413 cases). Some outdated input (e.g., authentication mechanism) are deprecated thus developers remove their support. For example, in commit ad4f1493 of octokit/rest.js, the support for `netrc` authentication method was removed while others were retained.
- To better organize the project (35 cases). Moving part of the functionality out and create a new package will make the project better organized. For example, in commit 0fa2830[23] of jsdoc/jsdoc, the package @jsdoc/core was reorganized and some methods in @jsdoc/core moved to a new package @jsdoc/cli.

Reason 2: To improve identifier names. (111 cases) Good identifier names can indicate the functionalities. However, improper identifier names can be a main contributor to technical debt in JavaScript projects. Improper identifier names may come from lack of familiarity with external knowledge and a failure to consider previous code. This reason can be divided into the following categories:

[23]https://github.com/jsdoc/jsdoc/commit/0fa2830

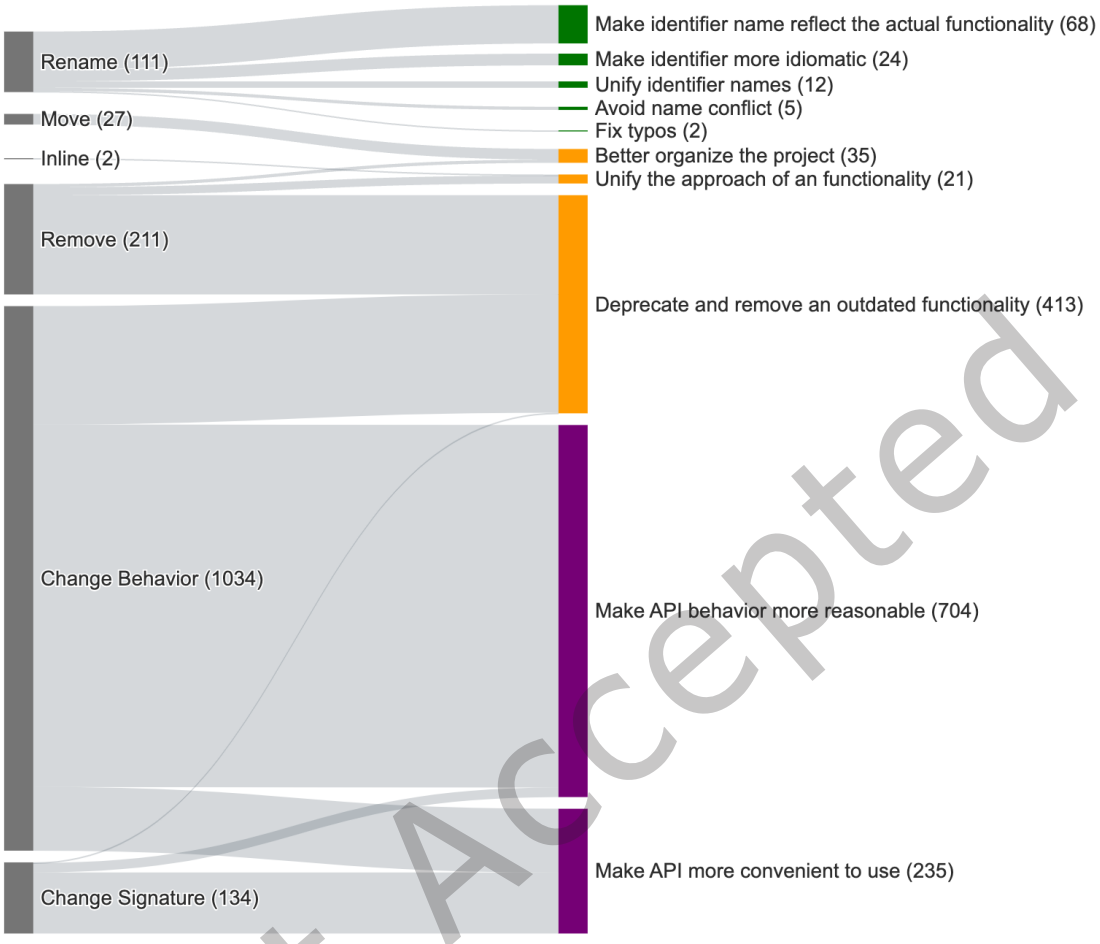


Fig. 3. Distribution of Motivations in Different Types of Breaking Changes

- To make the identifier name reflect the actual functionality (68 cases). The name of a class, interface or method, etc., should reflect the actual functionality. Therefore, after a code change, the identifier names tend to be modified. For example, in issue #3403 of angular/angular[24], the developer complaint that "When I first saw the requestCheck() method on ChangeDetectorRef, I assumed it was how I manually run change detection. Others have assumed this as well". Hence after the issue was reported, the identifier `requestCheck` is changed to `markForCheck`.
- To make the identifier more idiomatic (24 cases). For example, adding prefix `is` or `get` to the original name (e.g., from `empty` to `isEmpty`), and changing the order of tokens in an identifier (e.g., from `listSecretsForRepo` to `listRepoSecrets`).

[24]https://github.com/angular/angular/issue/3403

- To unify identifier names (12 cases). Similar classes or methods (e.g., components with the same style in a package) should share a similar name. For example, in commit 9bbd2469 of pnpm/pnpm[25], the field `localPackages` was renamed to `workspacePackages`. In the dependency @pnpm/resolver-base[26], `LocalPackages` had been renamed to `WorkspacePackages`, hence to make it consistent, the field was also renamed.
- To avoid name conflict (5 cases). For example, in commit da19583 of adonisjs/adonis-framework[27], developers renamed `Context` to `HttpContext` since `Context` is a commonly used keyword.
- To fix typos (2 cases). For example, renaming `MetricAarmProps` to `MetricAlarmProps`.

We also compared the rename operations in JavaScript (and TypeScript) with those in Java since most Java and JavaScript projects adopt the *camelCase* naming convention. For JavaScript and TypeScript projects, we show the rename operations in Table 4. For Java projects in Maven ecosystem, we inspect Huang et al.'s dataset on Java API migration [29]. Interestingly, we find that most method rename BCs are trivial, i.e., only adding or removing prefixes `get`, `is` (e.g., from `failed` to `isFailed`), and only few classes are renamed with one token replaced added, removed or replaced. Other popular languages like Python, Go and Rust, do not use *camelCase* convention.

Reason 3: To improve API design. (939 cases) The quality of API can impact the productivity of programmers, the adoption of APIs, and the quality of dependent code [44]. This reason can be divided into the following sub-categories:

- To make API behavior more reasonable (704 cases). The major scenarios of API's unreasonable behavior include: ① The API design does not conform to a certain standard and users may misuse the API. For example, in the commit 201f189d of angular/angular, after the code change, status codes less than 200 and greater than 299 will cause error, while previously, errors only occurred when network errors. ② The API behavior can easily cause unintended results. For example, in commit 7d4c399 of gajus/eslint-plugin-jsdoc, the behavior is changed unless a new option is set to true. This is to decrease the false positives when capitalized letters on newlines merely represent proper nouns.
- To make APIs more convenient to use (235 cases). For example, changing the parameter order of a method can be more in line with users' habits, and using `async` rather than `Promise` or callback parameters also provides convenience for downstream developers.

5 IMPLICATIONS

On the basis of the results above, we can yield the following implications for future works:
Automatic naming and renaming techniques can be applied in NPM projects. In Section 4.3, we conclude that the main reason for renaming identifiers is *to make identifier name reflect the actual functionality*. Therefore, on the one hand, poor identifier naming is a noticeable technical debt in JavaScript projects, hence automatic naming can be utilized to provide good identifier names, and on the other hand, in some cases, renaming identifiers is reasonable due to the necessary adaptation to frequent code changes in NPM ecosystem, thus we suggest applying automatic renaming techniques to make identifier names adapt to code evolution.

Several rule-based, static analysis-based, and machine learning-based approaches are proposed to help automatic naming and renaming. For example, the tool proposed by Caprile et al. [17] is a typical rule-based approach that makes unified tokens in an identifier comply to syntax rules, the tool proposed by Feldthaus et al. [27] is a static analysis-based approach to semi-automatically rename object properties with a focus on related property

[25]https://github.com/pnpm/pnpm/commit/9bbd2469
[26]This package is part of pnpm, and the Rename BC is located in the package @pnpm/find-workspace-packages, also a part of pnpm.
[27]https://github.com/adonisjs/adonis-framework/commit/da19583

identifiers[28], and RefBERT [37] is a typical deep learning-based approach that utilizes fune-tuning pre-training model (i.e., RoBERTa) and infers *local variable* names with contextual token sequences. However, these approaches suffer from two major limitations, and the future works can try to mitigate them:

- They are not proposed to deal with identifiers that can be directly accessed by downstream developers (e.g., class and interface names), while Rename actions on these identifiers make up the most in JavaScript projects. For instance, the most recent renaming approach RefBERT works in the same way as code completion, i.e., the intra-function context is input to infer local variable names, however, inferring identifier names needs different context knowledge, such as class structure, client invocation code, etc.
- They are not code-change-aware. Since we have found that many a proportion of Rename breaking changes are performed to ensure code consistency (Figure 3 in Section 4.3), a renaming technique had better leverage knowledge in prior related code changes to recommend proper names.

Tools for detecting code with similar functionalities are necessary. According to Figure 3, a number of Remove breaking changes are motivated by approach for unification, i.e., provide a unified API for a specific functionality. However, the relationship between these two or more code changes is indirect and not simple to detect. Besides, in our BC extraction process, we find developers make other small changes for better performance, compatibility, and robustness though they are unrelated to breaking changes. This also to some degree interferes with BC identification. The findings in our investigation can provide the following directions for future research:

- Investigate how to indicate potential BC actions when one BC has been already performed and recommend similar BC actions to improve code consistency and reduce code redundancy.
- Identify functionally related code snippets that can be removed together to facilitate clean code.
- Distinguish non-breaking changes from breaking changes and ordinary code changes to provide more convenience for developers.

Future research can focus on detecting more types of behavioral breaking changes. Existing approaches for JavaScript behavioral breaking change detection are NoRegrets [40] and its enhanced version NoRegrets+ [41], to the best of our knowledge. In Section 1, we have highlighted their reliance on test cases in downstream projects. In addition, the two approaches can only detect type-related breaking changes. Specifically, in the first phase, they run client test cases and monitor the flows of values by program instrumentation to build API models (intuitively, type restriction of APIs), and in the second phase, they rerun the client test cases with updated upstream code to check whether type restrictions are changed during execution. Therefore, they cannot detect non-type-related breaking changes, e.g., a BC that changes the semantics of the returned string, while the return type is still string. According to Section 4.3, there are four types of behavioral BC actions, where only *changing the specifications of return values* can directly cause type changes, while other behavioral changes may not change object types during execution. Therefore, we recommend future works try to deal with various types of behavioral breaking changes uncovered by this study.

6 THREATS TO VALIDITY

Threats to internal validity are mainly related to the clarity of developers' documentation of breaking changes. Few developers might have not explicitly documented the breaking changes in commit messages, and commit messages (as well as text in issues and pull requests) might not fully reflect developers' reason for making breaking changes. Also, the discussion and documentation related to breaking changes may not fully reflect developers' consideration. Second, in our manual analysis, we might still not understand developers' intentions since many breaking changes' commit messages do not clearly explain what and why they make breaking changes.

[28]Property identifiers refer to property names of an object, e.g., if `option` is a parameter, in `options.followSymLink`, `followSymLink` is a property identifier.

Additionally, we might regard some changes in code *diff* as BC-related actions since they are confusing. To address this potential threat, we have double-checked the analysis results to ensure the quality.

Threats to construct validity are concerned with the errors during breaking commit selection. In our study, we mainly consider the commits that follow Conventional Commits specification and collect breaking change commits that contain BREAKING CHANGE token, which may miss the commits that not fully follow the specification but are actually breaking. To mitigate this, we try our best to collect breaking changes from software documentation, especially *changelogs*, issues and pull requests (mentioned in Section 3.1). Additionally, we might overlook some commits, although they satisfy our selection criteria although we carefully inspect each collected commits (in Section 3.3).

Threats to external validity refer to the generalizability of our study. In our study, while we involve 1,519 breaking changes extracted from 381 randomly sampled projects from a vast number of popular NPM projects, and these projects cover diverse application fields of JavaScript language, such as utilities, frontend, Web APIs, database, etc., the extracted breaking changes only account for a small proportion of breaking changes in the whole NPM ecosystem. Besides, in this study we do not consider less popular projects since they have short development history and it is less probable to yield some patterns from these projects. However, the BC practices in less popular projects may still differ from those in popular projects. As for another threat, the Reid et al.'s dataset [48] used in our study was released in 2020 (four years ago), and might not include popular JavaScript and TypeScript released after 2020. We manually checked 500 NPM libraries with the largest number of dependents and found that between them[29] only 16 were initially published after 2020. Hence Reid et al.'s dataset probably covers most popular JavaScript and TypeScript projects, and this threat might not affect the generalizability too much.

Threats to conclusion validity relate to insufficient high-quality BC commits to support our findings. To mitigate the threats, we try our best to extract breaking changes from commit messages, issues, pull requests and *changelogs* to construct our breaking change dataset. We also carefully inspect our collected breaking changes and retain those related to JavaScript source code changes and explicitly documented by developers, which can help increase the quality of our dataset.

7 RELATED WORK

7.1 Research on NPM Ecosystem

A number of research works have investigated various aspects of the NPM ecosystem, which can help us understand the challenges in NPM ecosystem and provide future research directions. Decan et al. [23] studied the impact of vulnerabilities in the NPM ecosystem by analyzing how and when security vulnerabilities are reported and fixed, and to which extent they affect other packages in the NPM ecosystem in the presence of dependency constraints. Liu et al. [36] employed a knowledge graph method to characterize vulnerability propagation and evolution in the NPM ecosystem. Cogo et al. [20] conducted a systematic investigation into NPM dependency downgrades. Mujahid et al. [43] summarized the characteristics of highly-selected projects in NPM. Maeprasart et al. [39] illustrated the role of external pull requests in NPM project development. Abdalkareem et al. [12] examined the usage of trivial packages, while Chen et al. [19] further uncovered the motivations driving JavaScript developers to publish such packages despite their potential drawbacks. Venturini et al. [54] studied the impact of manifesting breaking changes in NPM, which is the most closely related work to ours. In contrast to their study, our research delves into multiple dimensions of breaking changes, including the affected program elements, the impact on client applications, and the underlying reasons behind these changes. Additionally, we consider the specific language features of JavaScript and TypeScript at a fine-grained level.

[29]We use the list in https://leodog896.github.io/npm-rank/PACKAGES.html, which is updated automatically (We accessed the page in September 2024).

7.2 JavaScript Static Analysis

Static analysis is a straightforward way to determine many properties of JavaScript programs and can be used to detect potential bugs and breaking API changes. Some open source tools like ESLint [6] and JSHint [8] support rule-based static analysis for JavaScript projects. They can be used for better code quality, such as making code follow JavaScript programming idioms. Some works proposed static analysis approaches from many aspects. For example, Madsen et al. built event-based call graphs [38] to detect event-related bugs by enhancing the static analysis framework JASI [33] and TAJS [32]. Other works [32, 45, 50] studied static analysis for JavaScript in the DOM environment. Furthermore, in addressing the limitations of static analysis approaches for JavaScript, Chakraborty et al. presented a technique to supplement missing edges in the JavaScript call graph [18]. Through their research, they discovered that dynamic property access is a primary factor contributing to low recall in previous static analysis frameworks, typically missing some function invocations. By applying their proposed technique, they were able to improve the recall rate. However, due to the complexity of language features, a static analysis approach cannot cover all language use cases. Also, these static analysis approaches focus on the detection and elimination of ill-formed code, and optimization of the testing process, rather than breaking changes among commits. By applying their proposed technique, they were able to improve the recall rate.

7.3 Breaking Change Detection and Analysis on Other Platforms

Besides JavaScript, a lot of studies focus on detecting and analyzing breaking changes for other programming languages, especially Java and Python, and their techniques may be learned by breaking change detection for JavaScript. Brito et al. presented APIDiff [16] that can detect syntax-related breaking changes in Maven projects, such as method removal and visibility loss by reusing the refactoring detection tool RefDiff [49]. Some open-source tools can also check Java syntactic breaking changes, such as Clirr [1] and RevAPI [10]. For Python language, Du et al. proposed AexPy [26] that can detect similar types of breaking changes like module removal and addition of required parameters, which extends the existing tool PyCompat [57] and Pidiff [9]. Regarding non-syntactic breaking changes, to the best of our knowledge, Zhang et al. proposed Sembid [56] to detect behavioral breaking changes by measuring the semantic difference of call graphs between old and new programs: if a code change's semantic difference is larger than a threshold and not identified as benign change, it is classified as a BC. However, their used semantic diffs reflect the structural changes of programs, while many semantic breaking changes do not need many of those changes, e.g., only changing one condition, and adding a small change to the output string. Additionally, as the authors have pointed out, Sembid cannot distinguish semantic breaking changes from non-breaking changes such as re-implementation of a method. Therefore, directly adapting Sembid to JavaScript projects is not suitable.

Additionally, with the help of breaking change detection techniques, many works analyzed the impacts of breaking changes to downstream projects. For example, Jayasuriya et al. [30, 31] investigated the impacts of breaking changes (especially behavioral breaking changes) to client applications in the Maven ecosystem. They concluded with many findings, e.g., 11.58% of the dependency updates contain breaking changes that impact the clients and 2.30% version updates have behavioral breaking changes that impacted client tests. Similar to the findings in Venturini et al.'s study [54], while the breaking changes are only detected in a small number of code changes or version updates, they still cause negative impacts to downstream projects.

8 CONCLUSION AND FUTURE WORK

In this study, for better comprehension of breaking changes in the NPM ecosystem and enhancing breaking change-related tools, we conducted an empirical study to bridge the knowledge gap from three aspects, with our carefully constructed breaking change dataset (1,519 breaking changes in total) sampled from a large set of popular NPM projects. We found that 95.4% of the breaking changes detected by regression testing can be

covered by developers' documentation, which proves that extracting breaking changes from documentation is reasonable. We then summarized the breaking changes in the NPM projects that are specific to JavaScript and TypeScript, and how developers make behavioral breaking changes, which yield many findings. Besides, we also investigated the reasons behind breaking changes in JavaScript and conclude with a taxonomy, which extend the previous works on motivations behind breaking changes. Based on our empirical findings, we provided actionable implications for future research, e.g., applying automatic renaming and naming techniques in JavaScript projects, and detecting code with similar functionalities, etc.

In the future, we want to collect additional sources such as online discussions, and employ alternative approaches like dynamic analysis to gain a deeper understanding of breaking changes in the NPM ecosystem. We plan to investigate how to automatically identify the breaking changes in the NPM projects. We consider combining static analysis and dynamic analysis techniques to enhance the existing breaking change detection approaches. We also consider improving breaking change detection with the help of large language models since they show great performance improvements in many software engineering related tasks and can utilize the semantics in source code.

ACKNOWLEDGMENTS

This research/project is supported by the National Science Foundation of China (No.62372398, No.72342025, and U20A20173), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] [n. d.]. Clirr. https://clirr.sourceforge.net.
[2] [n. d.]. Conventional Commits. https://www.conventionalcommits.org/en/v1.0.0.
[3] [n. d.]. Destructuring assignment. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment.
[4] [n. d.]. dont-break. https://www.npmjs.com/package/dont-break.
[5] [n. d.]. ECMAScript 2015. https://262.ecma-international.org/6.0/.
[6] [n. d.]. ESLint. https://eslint.org.
[7] [n. d.]. Inheritance and the prototype chain. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.
[8] [n. d.]. JSHint. https://jshint.com.
[9] [n. d.]. PiDiff. https://github.com/rohanpm/pidiff.
[10] [n. d.]. RevAPI. https://revapi.org.
[11] [n. d.]. Semantic Versioning. https://semver.org.
[12] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 385–395.
[13] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.
[14] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–56.
[15] Aline Brito, Marco Tulio Valente, Laerte Xavier, and Andre Hora. 2020. You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering* 25 (2020), 1458–1492.
[16] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 507–511.
[17] Caprile and Tonella. 2000. Restructuring program identifier names. In *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 97–107.

[18] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. 2022. Automatic root cause quantification for missing edges in javascript call graphs. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[19] Xiaowei Chen, Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Xin Xia. 2021. Helping or not helping? Why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering* 26 (2021), 1–24.
[20] Filipe Roseiro Cogo, Gustavo A Oliva, and Ahmed E Hassan. 2019. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2457–2470.
[21] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*. IEEE, 275–284.
[22] Alexandre Decan and Tom Mens. 2019. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* 47, 6 (2019), 1226–1240.
[23] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.
[24] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.
[25] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.
[26] Xingliang Du and Jun Ma. 2022. AexPy: Detecting API Breaking Changes in Python Packages. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 470–481.
[27] Asger Feldthaus and Anders Møller. 2013. Semi-Automatic Rename Refactoring for JavaScript. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications*. 323–338.
[28] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. A large-scale empirical study on Java library migrations: prevalence, trends, and rationales. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 478–490.
[29] Kaifeng Huang, Bihuan Chen, Linghao Pan, Shuai Wu, and Xin Peng. 2021. REPFINDER: Finding replacements for missing APIs in library update. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 266–278.
[30] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1238–1261.
[31] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1433–1444.
[32] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. 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*. 59–69.
[33] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 121–132.
[34] Dino Konstantopoulos, John Marien, Mike Pinkerton, and Eric Braude. 2009. Best principles in the design of shared software. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 2. IEEE, 287–292.
[35] Meir M Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.
[36] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 672–684.
[37] Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. Refbert: A two-stage pre-trained framework for automatic rename refactoring. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 740–752.
[38] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node. js JavaScript applications. *ACM SIGPLAN Notices* 50, 10 (2015), 505–519.
[39] Vittunyuta Maeprasart, Supatsara Wattanakriengkrai, Raula Gaikovina Kula, Christoph Treude, and Kenichi Matsumoto. 2023. Understanding the role of external pull requests in the NPM ecosystem. *Empirical Software Engineering* 28, 4 (2023), 1–23.
[40] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *32nd european conference on object-oriented programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[41] Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node. js libraries. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 409–419.
[42] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*. 215–225.

[43] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. 2023. What are the characteristics of highly-selected packages? A case study on the npm ecosystem. *Journal of Systems and Software* 198 (2023), 111588.
[44] Brad A Myers and Jeffrey Stylos. 2016. Improving API usability. *Commun. ACM* 59, 6 (2016), 62–69.
[45] Changhee Park, Sooncheol Won, Joonho Jin, and Sukyoung Ryu. 2015. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 552–562. https://doi.org/10.1109/ASE.2015.27
[46] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 378–387.
[47] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.
[48] Brittany Reid. 2020. *NPM Package Information from Libraries.io.* https://doi.org/10.5281/zenodo.3898749
[49] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.
[50] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. 2016. Static DOM event dependency analysis for testing web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 447–459.
[51] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International symposium on the foundations of software engineering*. 1–11.
[52] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message?. In *Proceedings of the 44th International Conference on Software Engineering*. 2389–2401.
[53] Luca Traini, Daniele Di Pompeo, Michele Tucci, Bin Lin, Simone Scalabrino, Gabriele Bavota, Michele Lanza, Rocco Oliveto, and Vittorio Cortellessa. 2021. How software refactoring impacts execution time. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2021), 1–23.
[54] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A Gerosa, and Igor Scaliante Wiese. 2023. I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–26.
[55] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 138–147.
[56] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has my release disobeyed semantic versioning? Static detection based on semantic differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[57] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How do python framework apis evolve? an exploratory study. In *2020 ieee 27th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 81–92.

Received 17 April 2024; revised 3 October 2024; accepted 11 October 2024