Where to Go Now? Finding Alternatives for Declining Packages in the npm Ecosystem

Suhaib Mujahid
Mozilla Corporation
Montreal, Canada
smujahid@mozilla.com

Diego Elias Costa
Université du Québec à Montréal
Montreal, Canada
costa.diego@uqam.ca

Rabe Abdalkareem
Omar Al-Mukhtar University
Bayda, Libya
rabe.abdalkareem@omu.edu.ly

Emad Shihab
Concordia University
Montreal, Canada
emad.shihab@concordia.ca

Abstract—**Software ecosystems (e.g., npm, PyPI) are the backbone of modern software developments. Developers add new packages to ecosystems every day to solve new problems or provide alternative solutions, causing obsolete packages to decline in their importance to the community. Packages in decline are reused less over time and may become less frequently maintained. Thus, developers usually migrate their dependencies to better alternatives. Replacing packages in decline with better alternatives requires time and effort by developers to identify packages that need to be replaced, find the alternatives, asset migration benefits, and finally, perform the migration.**

This paper proposes an approach that automatically identifies packages that need to be replaced and finds their alternatives supported with real-world examples of open source projects performing the suggested migrations. At its core, our approach relies on the dependency migration patterns performed in the ecosystem to suggest migrations to other developers. We evaluated our approach on the npm ecosystem and found that 96% of the suggested alternatives are accurate. Furthermore, by surveying expert JavaScript developers, 67% of them indicate that they will use our suggested alternative packages in their future projects.

Index Terms—**Dependency Suggestions, Dependency Quality, Package in decline, Dependency, npm, JavaScript.**

I. Introduction

Software ecosystems such as npm, Maven, and PyPI save us from reinventing the wheel over and over by facilitating the reuse of code to implement common functionalities [1]. For example, the registry of the node package manager (npm) alone hosts more than 2.3 million packages to date and is still growing exponentially [2]. Leveraging packages from the ecosystem can boost development productivity [3], and improve software quality [4]. However, there is no such thing as a free lunch. The large size and rapid increase in the number of packages in the ecosystem have drawbacks. Developers need to spend time and effort to find the right package to use in their projects. Also, since software, like people, gets old [5], developers need to keep up with the changes in the ecosystem to avoid depending on packages that became obsolete, dormant, or even deprecated [6].

Community interest uphold packages to improve, i.e., include better features driven by community needs, keep up the package maintenance by reporting bugs to maintainers, motivate maintainers to continue supporting the package, and some times even financially support the maintainers on platforms such as GitHub Sponsors [7] and Open Collective [8]. Packages that show a decline in community interest are usually used less over time, become less frequently maintained, and eventually, could become abandoned [9, 6]. Moreover, a package's decline in community interest may indicate that a better solution is drawing attention in the ecosystem, and developers are migrating to a package that better suits their needs [10].

Prior work examined projects that are unmaintained [11, 12] and identified packages that lose popularity over time (i.e., are in decline) [10]. Other studies proposed approaches for mining dependency migrations from software repositories to suggest alternatives [13, 14, 15, 16]. However, to the best of our knowledge, little attention has focused on suggesting alternatives to packages that are in decline, especially in the context of dynamic programming languages such as JavaScript.

Therefore, in this study we leverage the wisdom of the crowd in the software ecosystem to suggest alternatives to packages that are in decline. Our approach uses dependency migrations from real-world projects to identify alternative packages. Moreover, our approach suggests dependency migrations based on the community interest of the packages, by replacing the packages that are in decline with alternative packages that still maintain the community interest.

We evaluate our approach on the npm ecosystem, the host of JavaScript reusable packages and the largest and most popular ecosystem to date [17, 18]. The popularity and scale of the npm ecosystem make it an ideal candidate for our study. We evaluate the accuracy and the usefulness of our approach in generating alternatives for packages in decline, through the following three research questions:

RQ1: How accurate is our approach in suggesting alternative npm packages? (Section III) Our approach identified 152 dependency migration suggestions, of which 96% are valid suggestions. Suggestions include alternatives of 10+ different package categories, from packages that assist project builds to implementing user interfaces, showing the wide applicability of our approach.

RQ2: When and why maintainers migrate to depend on the alternative npm packages? (Section IV) The majority of migrations (74%) are primarily motivated to replace unmaintained dependencies, which could cause maintenance issues for software projects. Most migrations occur as a dedicated maintenance task (69%), however, 31% of migrations occur during other development tasks, i.e., fixing bugs (16%), adding new features (8%) and code refactoring (7%).

RQ3: How useful is our approach to JavaScript project maintainers? (Section V) We surveyed 52 JavaScript developers to assess the usefulness of our approach. We found that our approach provided new information about alternative packages for 54% of the developers. On a 5-points Likert scale, developers recommend having a tool that utilizes our approach to suggest alternative packages with median = 4. More importantly, 67% of the developers confirmed that they would use our suggested alternative packages in their future projects.

Our findings show that our approach is accurate and helpful to JavaScript developers. The following are the key contributions of our paper:

- Propose an approach to suggest alternatives for packages in decline using migration trends in the software ecosystem.
- Empirically evaluate our approach accuracy on the npm ecosystem, and investigate the characteristics of the dependency migrations suggested by our approach.
- Surveyed 52 expert JavaScript practitioners to assess the usefulness of our approach through the awareness of suggestions, perception of usefulness, and whether our tool would motivate action from practitioners.
- Support the replication and future research by making all of our datasets (i.e., collected data, analysis results, scripts) publicly available [19].

II. Approach

In this section, we explain our approach that uses the dependency migration patterns in the npm ecosystem and the packages' centrality trends to suggest alternative packages for the ones that are in decline. Figure 1 shows the overall workflow of our approach, which is further detailed in the remainder of this section.

A. Detect Dependency Change Events

The core of our approach lies in identifying frequent dependency migration patterns occurring in the ecosystem to better inform or recommend practitioners. To extract dependency migration patterns or compute the centrality trends, we need to detect *dependency change events* in the npm ecosystem. In our study, we consider two events as dependency change events: 1) the addition of a new package dependency and 2) the removal of a package dependency. Hence, dependency change events do not include updating the version of a dependency, since our approach aims to suggest dependency migrations regardless of their versions. Also, we use the dependency change events to update the npm ecosystem's dependency graph and calculate the centrality trends (in Section II-C).

To extract dependency change events for all packages in the npm ecosystem, we analyze the entire npm registry database. The npm registry maintains a record of the packages' dependencies for each version of every package. For each package in the registry, we start by sorting the package versions in ascending order by their release time. Then, on each version (v_n), we compare the list of dependencies with the previous version (v_{n-1}). If the dependency is absent in the version v_{n-1}, we consider it to be a dependency addition event; conversely, if the dependency is absent in the version v_n, we consider it a dependency removal event.

It is crucial to note that package releases can be nonlinear. Package maintainers commonly employ backports to fix older release versions [20]. In such cases, the chronological order of the versions will be polluted by backports, as these versions could include old dependencies no longer used in the main release branch. Hence, in our process, we filter out any release with a lower semantic versioning than its predecessor in relation to their respective release date [10]. For example, the developers of the package `react` have released the version `16.13.1` in March 2020, then in October 2020 released the backport version `15.7.0` to fix an older major version, where the latest version was `15.6.2` [21]. Since the version `15.7.0` is smaller than the version `16.13.1`, we exclude the version `15.7.0` from our analysis.

B. Extract Dependency Migration Patterns

We extract dependency migration patterns by identifying recurring *dependency replacements* in the npm ecosystems, where a dependency gets dropped in favour of an alternative package that performs the required functionalities. First, we use the dependency change events to retrieve changes in the packages' dependencies across all their versions. Then, we consider each of the added dependencies as a potential replacement for each of the removed dependencies. Listing 1 show an example of dependency changes in a `package.json` file between two package versions. The dependency changes in the example are represented as four dependency change events. Three dependencies are removed (i.e., removal change events), and one dependency is added (i.e., an addition change event). By applying this on the example in Listing 1, we extract three dependency replacements: `less` $\rightarrow$ `lodash`, `underscore` $\rightarrow$ `lodash` and `utf-8-validate` $\rightarrow$ `lodash`. In our process, we do not mix runtime dependencies with development dependencies. Thus, we consider the added runtime dependencies as potential replacements for only the removed runtime dependencies, not the development dependencies, and vice versa. We do this since the development and runtime dependencies are typically used in different contexts and should not be recommended as alternatives to each other.

When a package releases a new version and replaces more than one dependency, our approach has no way to identify which dependency has been replaced nor its replacement. Thus, as explained earlier, we consider each added dependency a potential replacement for each deleted dependency (combination). Since we want to reduce the odds of combinatorial explosion of dependency replacements, we filter out releases with massive or imbalanced number of dependency changes. We only consider dependency change events from releases where the difference between the number of added dependencies (D_a) and removed dependencies (D_r) are close,

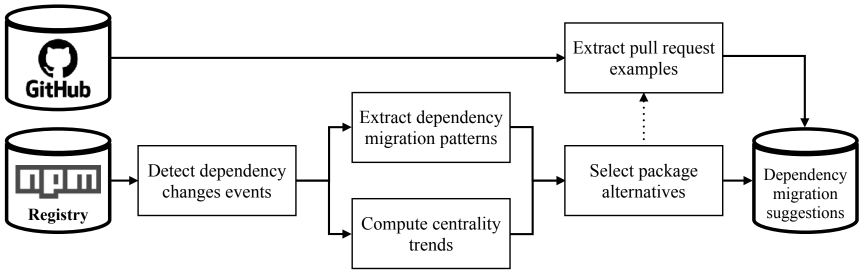
GitHub
npm
Registry
Detect dependency changes events
Extract dependency migration patterns
Compute centrality trends
Extract pull request examples
Select package alternatives
Dependency migration suggestions

Figure 1: Our approach to suggest package alternatives.

Listing 1: Dependency changes taken from the version `0.2.16` of the package `jpmorganchase/perspective`.

```
    "dependencies": {
        "detectie": "1.0.0",
        "flatbuffers": "^1.10.2",
-       "less": "^2.7.2",
+       "lodash": "^4.17.4",
        "moment": "^2.19.1",
        "tslib": "^1.9.3",
-       "underscore": "^1.8.3",
-       "utf-8-validate": "~4.0.0",
        "websocket-heartbeat-js": "^1.0.7",
        "ws": "^6.1.2"
    },
```

i.e., $|D_a - D_r| \leq 1$. Also, we avoid considering change events where there is a large number of added and removed dependencies, i.e., $D_a + D_r \leq \tilde{x}$, where $\tilde{x}$ is the median value for $D_a + D_r$ across all the releases in the npm registry. We do this filtering since a large number of dependency changes indicates a significant code refactoring more than a simple dependency replacement.

We consider a *dependency migration pattern*, a dependency replacement that frequently occurs in the ecosystem, which is more likely to indicate a trend. Hence, after identifying the dependency replacements, we consider replacements that re-occur at least 10 times in our dataset as dependency migration patterns. That is, at least the developers of 10 distinct projects must have performed the same dependency replacement.

C. Calculate Centrality Trends

Centrality has been used as a proxy of community interest, where packages that show a decline in centrality are usually used less over time, become less frequently maintained, and eventually could become abandoned [10]. Thus, our approach uses the centrality (i.e., measured using PageRank [22]) to target suggesting alternatives for packages that are in decline to replace them with packages deemed not in decline. To determine if a package is in decline or not, we use the approach proposed by Mujahid et al. [10] which requires dependency change events to calculate the centrality and detect packages in decline. A package considered in decline if it shows statistically significant declines in the centrality over a specific period of time.

Our approach requires the centrality trends for packages engaged in the extracted dependency migration patterns. However, calculating the centrality rankings requires computing the centrality for every package in the ecosystem. Thus, we use the dependency change events (extracted in Section II-A) to calculate the monthly centrality rankings for each package in the npm registry.

D. Select Package Alternatives

Once we have both the dependency migration patterns (Section II-B) and centrality trends for all packages (Section II-C), we select the most promising dependency migration patterns to recommend to practitioners. To do so, we use the following criteria:

- **The replaced package is in decline:** we select only patterns where the removed package is in decline. Since the decline can vary based on the examined period, we measure the decline over three different periods: the last six months, the last year, and the package's overall lifetime. If the package shows a decline based on one of the measured periods, we consider it an in decline package.
- **The alternative package is not in decline:** to ensure that we suggest better alternatives, we select only patterns where the added package is not in decline.
- **The migration pattern is performed recently:** to avoid recommending outdated migration patterns, we consider only the patterns performed at least once in the last 90 days.
- **Performed by a popular project:** to avoid considering patterns performed only by immature projects, a migration pattern should be performed by a popular project in order to be considered. In this context, we consider a project as popular if the project is in the top 10% of most central packages in the npm ecosystem.

When our approach finds more than one package alternative, we select the dependency migration pattern with the highest support, i.e., performed more frequently.

E. Extract Pull Request Examples

We aim to provide developers with examples of pull requests that performed the suggested dependency migration. Exemplary pull requests may provide insights on the migration's efforts, reasons for the dependency migration, and help practitioners understand the differences between the alternatives. To extract pull requests examples, we check the npm registry to find packages that performed any of the candidate dependency migration patterns. For these packages, we collect their repository addresses on GitHub. Then, we use the GitHub GraphQL API [23] to extract all the merged pull requests from the selected repositories.

Once we retrieve the pull requests from a repository using the GitHub API, we select only the pull requests that perform the suggested dependency migrations. To do so, we consider only the pull requests that modify a `package.json` file, which is the file where projects declare their dependencies. Then, we compare the content of the `package.json` file as it is on the merge commit with the content of the files as it was on the parent commit. Next, we exclude pull requests that are extremely big, i.e., change more than 100 files, which is 7 times more that average number (mean = 13.62) of changed files in pull requests [24].

III. Accuracy of the Approach

The decline of package centrality is a symptom that better alternatives have emerged, shifting the community interest. However, developers have little information to grasp where the community has shifted its interest [10]. If our approach can effectively capture valid package alternatives, it can be embedded in dependency management tools, such as the npm CLI, to increase developers' awareness of alternative packages and help them reevaluate their package dependencies.

A. Approach.

To measure the accuracy of our approach, we first use it to generate dependency migration suggestions to alternative packages. Then, we manually evaluate whether the suggested alternative packages perform comparable functionalities to the original ones.

Generate Dependency Migration Suggestions. We generate the suggestions to alternative packages using the approach described in Section II. We start by detecting the dependency change events as of December 22, 2020. We collected in total 18,459,923 dependency change events from 1,148,720 packages from the npm ecosystem. From these change events, we extracted 2,434 dependency migration patterns. After filtering the migration patterns based on the centrality trend of the packages and the criteria described in Section II-B, we end up with 152 package alternative suggestions.

Manual Evaluation. Once we generate the dependency migration suggestions, we manually evaluate the correctness of the suggested alternative packages by assessing whether both packages provide similar functionalities or not. We manually examine the documentation (e.g., readme file, homepage, and website) of the package to be replaced and the suggested alternative package to understand their functionalities. We start by inspecting the documentation of the package homepage on the official npm website [25]. If the description is not descriptive enough for our classification, we examine other available sources such as the readme file, the package website, and the package repository. Once we have a comprehensive understanding of both packages, if the suggested alternative package performs comparable functionalities to the original package, we consider the suggested alternative package as a valid alternative package. For example, the packages `commander` and `yargs` share similar functionalities, helping in building command-line interfaces for *Node.js*, thus, we consider them as valid alternative packages.

In total, two of the authors independently examined the documentations of 256 packages for 152 dependency migration suggestions. Since this process involves human judgment, it is prone to human bias. We assess the agreement of both examiners using the Cohen-Kappa inter-rater reliability. Cohen-kappa inter-rater reliability is a well-known statistical method that evaluates the inter-rater reliability agreement level. The result is a scale that ranges between -1.0 and 1.0, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement [26]. In our analysis, we found that both authors have an excellent agreement (kappa=0.79). After the initial classification, any disagreement was examined and discussed by both examiners to reach consensus [27].

B. Results

Based on the manual evaluation of our approach, Table I shows a summary of the migration suggestions generated by our approach. We categorize and group the suggestions by their abstracted functionalities. Each category of suggestions in the table has an example of a package migration suggestion generated by our approach. The third column presents the number of True Positive cases (TP), where our approach accurately finds the alternative packages. The last column shows the number of False Positive cases (FP), where our approach suggested invalided alternatives.

Table I: Summary of the suggested alternatives categories.

Category	Suggestion Example	TP	FP
Building	`browserify` → `webpack`	39	3
Utilities	`moment` → `dayjs`	26	-
Testing	`vows` → `mocha`	16	-
User Interface	`jade` → `pug`	12	-
Linter	`jshint` → `eslint`	10	2
Client Library	`redis` → `ioredis`	10	1
Networking	`request` → `axios`	10	-
Parser	`esprima` → `acorn`	9	-
CLI	`optimist` → `yargs`	4	-
Other	`memory-fs` → `memfs`	10	-
Total		**146**	**6**

Overall, we examine 152 dependency migration suggestions generated by our approach. We found that 146 (96%) of the generated dependency migration suggestions include valid alternative packages and 6 (4%) of them do not. Furthermore, we found that the packages performing functionality related to building the JavaScript projects have the highest share (28%) of the generated dependency migration suggestions. For example, our approach suggests replacing the JavaScript bundler package `browserify` with a more popular, scalable, and feature rich one, the `webpack` package. The next category in our results is the utility tools (17%), where our approach suggests replacing obsolete utility packages such as `moment` with a more modern solution like `dayjs`. Next categories include testing tools, and user interface components and helpers with share of 11% and 8% respectively. Also, among others, the categories include linters to enforce rules on the JavaScript code, clients drivers to interact with other services, and networking utilities.

Out of the 6 invalid dependency migration suggestions that we found, only one invalid suggestion belongs to replacing a runtime dependency, whereas the remaining 5 cases suggest replacing development dependencies. An example of invalid dependency migration suggestion is the suggestion of replacing the development dependency `ember-cli-eslint` with `eslint`. However, `ember-cli-eslint` is a plugin to identify and report patterns found in Ember projects [28], not a valid alternative package of `eslint`.

Summary of RQ1: Out of the 152 dependency migration suggestions generated by our approach, we found that 96% of them are valid alternative package suggestions. Most frequent suggestions recommend alternatives for building, utilities and testing packages.

IV. Characteristics of the Suggestions

We want to understand *why* and *when* developers do the kind of package dependency migrations that our approach suggests. Answering the question *why* will help us find the reasons developers migrate to the alternative packages and how their motivation is tied to software maintenance. Answering the question *when* aims to discover the types of maintenance activities involved in migrating dependencies. Both questions help us understand the reasons to use suggested migrations and how to employ our approach during software development.

A. Approach

To understand when and why developers migrate their dependencies, we manually classify pull requests that perform dependency migrations that match the migration suggestions by our approach. We start by extracting pull requests using the approach described in Section II-E. As a result, we obtain a list of 225 pull requests that perform dependency migrations from 155 different GitHub repositories.

Once we have the dependency migration pull requests, we perform an iterative coding process to classify and group pull requests. We gradually develop two sets of codes based on an inductive analysis approach [29]. The first set of codes concerned the purpose of the pull request (activity) and the second set of codes concerned the motivation of the dependency migration performed in the pull request.

In the process of classifying the activity of a pull request, we focus on the primary goal of the pull requests, as described in its title and description. Thus, we tag each pull request with only one activity type. Also, we examine the pull request description and discussion to find the motivation of the dependency migration. However, not all pull requests include an explanation to justify the dependency migration. For the ones that have a justification, we tag them with the migration motivation. Two authors independently tag each of the 225 pull requests with a single activity type, and applicable pull requests are also tagged with the migration motivation.

Table II: The motivations of 62 pull requests that performed the dependency migrations.

Motivation	Description	Frequency
Maintenance	Better quality and better maintained alternative.	74%
Compatibility	Increase compatibility with other packages or systems.	15%
Performance	Faster execution, less dependencies, or smaller bundle size.	8%
Features	Providing missed features or flexible API.	3%

Table III: The activities of the pull requests that performed the dependency migrations.

Activity	Description	Frequency
Dependency update	Intended mainly to update dependency versions.	42%
Dedicated migration	The main goal is to migrate to a different package.	27%
Bug fixing	Aims to resolve issues in the project.	16%
New feature	Adds a new functionality or feature to the project.	8%
Refactoring	The objective is refactor existing code.	7%

As with any other manual classification activity, there is some level of subjectivity that may generate disagreement between the annotators. To account for this, we applied a Cohen's Kappa to measure the level of agreement between the two individual classifications [30]. In our analysis, we found that both authors have an excellent agreement (kappa=0.93) on classifying the pull request activates. Also, the authors have an excellent agreement (kappa=0.90) on classifying the motivation of the dependency migrations.

B. Results

Based on the manual classification of the pull requests, we organize the results into two parts. The first part discusses the motivation for migrating dependencies, and the second part dives into the types of maintenance activities performed during migration.

Why developers migrate the dependencies? Our manual assessment found that 62 (24%) of the 225 pull requests provide an explicit justification of the dependency migration. In Table II, we show the results of our manual classification.

In 74% of the cases, dependency migrations are motivated by the need for better maintained alternative packages. Even when the package is not officially deprecated, developers migrate from packages that are not well maintained, for example [31]: "Removes isomorphic-fetch from the dependencies which doesn't seem to be maintained anymore." Even when a package has maintenance activities, developers were not satisfied with the quality level of the maintenance, one of the developers said [32]: "Sentry is moving away from the Raven library we are using and while it says it's maintained, not every feature is working anymore."

The second most frequent motivation for dependency migrations in our dataset is the compatibility with other packages or systems (15%). In these cases, developers migrate to use alternative packages that are more compatible with other project dependencies or to support more systems and platforms. For example, in one of the pull requests the th developer migrated from the package `uglifyjs` to the package `terser` to be compatible with the new version of `webpack` package [33]: "Webpack requires uglifyjs-webpack-plugin@1.x. thus uglifyjs-webpack-plugin@2.x may not resolve correctly. Also, the webpack team decided to go with terser-webpack-plugin." In another case, the migration performed to improve the support for the Windows operation system [34]: "It is a nightmare to run this project on Windows as it uses bcrypt."

We observe that 8% of the dependency migrations are motivated by improving the performance of the project. Performance metrics mentioned include faster execution time, smaller bundle size shipped to production, and lower number of transitive dependencies. One developers improved the performance by migrating from the package `moment` to the package `dayjs` [35]: "`dayjs`, after webpacking, is about 7 KB compared to about 700 KB for `moment`. This will mean faster load times and smaller packed VSIXs."

In the remaining of the pull requests (3%), we observe that the motivation is to use features offered by the alternative packages. For example, a developer migrated from `sanitize-html` to `dompurify` to allow for more HTML tags after sanitizing HTML content [36]: "`dompurify` prevents XSS but allows more tags and attributes than our previous sanitizer."

When developers perform dependency migrations? Table III shows our classification results of the pull request activities. We found that 42% of the pull requests perform dependency migrations as part of a dependency update. For example, in one of the projects, the developer created a pull request to update multiple dependencies, and in the same time migrated from using the package `node-uuid` to `uuid` [37]. Interestingly, we found that only 27% of the pull requests are dedicated to performing a dependency migration activity. In a dedicated migration, the main goal of the pull request is to just replace a dependency with another. In contrast, a pull request tagged with dependency update activity aims to update the version of one or more dependencies, but it replaces other dependencies in the same pull request.

From our analysis, we identified that 16% of the dependency migrations were a part of a bug fixing activity. For example, in one of the projects, a developer created a pull request to fix a bug by migrating to an alternative package [38]. He describes the issue as the following: "isomorphic-fetch has a bug that prevents it from running in a react native environment. Since it is no longer maintained, it will never be fixed. That also means dependencies are outdated. cross-fetch is React Native compatible." In 8% of the cases, we observe that the dependency migration occurs when developers add a new feature. An example of such a case, a developer migrated to an alternative package to support the out of the box installation on more platforms [34]: "And I am not arguing that it is not possible to install keystone on Windows, but this is far from a simple npm install. And as bcrypt is the problem, switching to bcryptjs would make it easier." In the remaining cases (6%), we notice that the dependency migration was a part of refactoring activity. For example, a developer refactored the code to use plain TypeScript and migrated from depending on the package `rimraf` to the package `del-cli`.

Summary of RQ2: The majority of dependency migrations (74%) are performed to replace unmaintained dependencies, followed by compatibility issues (15%) and performance problems (8%). Developers often migrate packages in dedicated dependency-related pull requests (69%), but also migrate while fixing bugs (16%), including new features (8%), and refactoring (7%).

V. Usefulness of the Approach

In this section, we evaluate the usefulness of our approach. We ask practitioners if they find our suggestions useful and practical for the maintenance of their software projects. More specifically, we want to know if our approach presents new information to practitioners: Are practitioners aware of the suggested migrations? Do practitioners believe our approach is valuable? Would practitioners act upon the suggested changes?

A. Approach

We conducted a survey to collect feedback from JavaScript project maintainers. In the following, we will present our survey design, participant recruitment process, the background of the participants, and the survey results.

Survey Design We design a survey to evaluate the usefulness of the suggested dependency migration to JavaScript developers that use the packages in decline in their projects. That is, we target developers that have used the packages our approach recommends replacing. Table IV shows the questions we ask along with the types of accepted answers for each question. Our survey contains three main parts, we ask JavaScript practitioners:

1) Their **software development background**, to ensure that our survey participants have sufficient experience using npm packages in software development and maintenance.
2) Their **awareness of suggested alternatives**, to assess whether our tool can be used to inform developers of migration patterns in the ecosystem.
3) Their **perceptions of our suggested dependency migrations**, to assess the degree of usefulness of our approach in a real scenario. We ask participants two groups of

Table IV: Questions in our survey about the alternative package suggestions.

Category	Question	Accepted Answers
Background	How would you best describe yourself?	*Single selection options:* Full-time, Part-time, Free-lancer, or Other.
	For how long you have been developing software?	*Single selection options:* Less than 1 year, 1 to 3 years, 4 to 5 years, or More than 5 years.
	How many years of JavaScript development experience do you have?	
	How many years of experience do you have using the Node Package Manager (npm)?	
	How often do you search for npm package alternatives?	*Single selection options:* Never, Rarely (once a year), Sometimes (once a month), Often (once a week), or Very often (everyday).
Awareness	Are you aware of the alternative package mentioned in the email?	*Single selection options:* Yes or No.
	Are you aware of the JavaScript community's migration trend mentioned in the email?	
Usefulness	Do you think that a tool that helps generate potential alternative packages would be useful?	*Likert-scale:* ranges from 1 = Not useful, to 5 = Extremely useful.
	Do you believe that providing an example of Pull Requests of migrations from other projects would be helpful?	*Multiple selection options:* Help in estimating the dependency migration efforts, Help in justifying the dependency migration, Help in understanding the required API changes, Not helpful, and Other.
Future Actions	In your future new projects, will you use the alternative package?	*Single selection options:* Yes or No.
	If the previous answer was "No", why not?	*Free text*
	In your current projects, will you advise to migrate to the alternative package?	*Likert-scale:* ranges from 1 = Keep the current package, to 5 = Strongly advise migrating.

questions: 1) how useful they found our suggested dependency migrations, and 2) their willingness to take actions related to our suggestions. In this section, we also included open-ended questions to give our survey participants the flexibility to express their opinion and experience, as recommended in survey design guidelines [39].

Participant Recruitment To select our survey participants, we reach out to experienced developers who have adopted the packages that we suggest being replaced with alternatives. We retrieve a list of JavaScript projects hosted on GitHub that have at least 100 stars, as commonly done in the related literature [40, 41, 42], to mitigate the chances of including too many immature and personal projects in our survey. We retrieve a list of 35,719 projects using the GitHub API [43]

Next, we use the GitHub raw content API [43] to retrieve the list of dependencies from the `package.json` file of each project. If a project has any dependencies that our approach suggests being replaced, we clone the project's repository to be analyzed. To select participants with experience in the target dependencies, we target the developers who introduced these dependencies in their projects. Thus, we use `git` to retrieve the change history of the `package.json` file. On each commit that modifies the `package.json` file, we detect the dependencies that were added, by comparing commit diffs. If the commit is adding a dependency that our approach suggests being replaced, we retrieve the author contacts from the commit metadata.

To prevent sending teams multiple survey invitations and to diversify our participants, we only select one developer per GitHub organization. Based on these steps, we identify 4,696 unique JavaScript developers. Then, we randomly selected 1,000 developers to participate in our survey. Finally, we send email invitations of our survey to 1,000 JavaScript developers and successfully reached to 886 developers (some emails were not delivered, e.g., email address not found). We received 52 responses for our survey in the first two weeks, leading to a 6% response rate, comparable to the response rate reported in other software engineering surveys [44, 45].

Survey Participants. Table V shows the background of our survey participants, including their position, their experience in software development, JavaScript development, and the use of the npm packages. Overall, the majority of participants work full-time (42 out of 52), and have at least 5 years of experience in software development in general (47 out of 52), JavaScript development (41 out of 52), and using npm (40 out of 52).

We also ask participants how often they search for alternative npm packages, to evaluate their interest and experience in finding better npm packages for their projects. Figure 2 show that out of the 52 participants, 92% of them do search for alternatives npm packages, consolidating this task as a common maintenance task in the maintenance of software projects. The frequency in which developers look for other packages varied from once a month (52%), once a year (29%), once a week (6%), and every day (6%). Interestingly, only 8% of our survey participants report that they never search for alternative packages.

B. Results

We measure our approach usefulness along three dimensions: 1) developers awareness of the generated suggestions,

Table V: Participants' position and experience in software development, JavaScript development, and using npm.

Developers' Position	Occurrences	Development Experience	Occurrences	Experience in JavaScript	Occurrences	Experience in Using npm	Occurrences
Full-time	42	1 - 3	0	1 - 3	3	1 - 3	3
Part-time	1	4 - 5	5	4 - 5	8	4 - 5	9
Freelancer	9	> 5	47	> 5	41	> 5	40

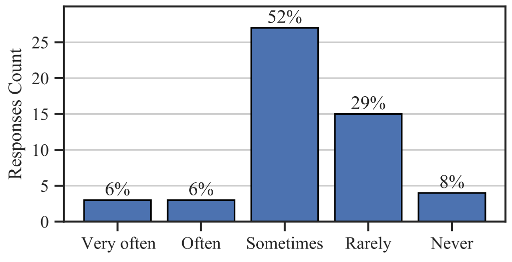

Figure 2: Survey responses to how often our would participants search for package alternatives. The question has the following answers: never, rarely (once a year), sometimes (once a month), often (once a week), very often (everyday).

2) developers perceptions of our suggestions, and 3) developers willingness to take future actions based on our suggestions.

Developer Awareness. We assess the awareness about the generated suggestions by asking the participants 1) if they know the alternative package and 2) whether they are aware of the migration trend in the JavaScript community toward the alternative packages. Based on our survey responses, our approach was able to inform participants about the alternative packages and the JavaScript community's migration trend. Specifically, Figure 3 shows that 37% of the participants are not aware of suggested alternative packages, and 48% of the participants are not aware of the dependency migration trend in the JavaScript community. Given our suggestions are based on migrations that have been performed many times in the npm ecosystem, we found it surprising that 37% of participants had not heard of the alternative package before. To put things into perspective, all of our survey participants are familiar with the original package and 92% have reported to regularly looking in the ecosystem for alternative packages. This indicates that even experienced developers are frequently unaware of the ecosystem' trends and need tools to be better informed about their community.

Developer Perceptions. To understand the perception of the project maintainers of our approach's results, we ask participants if they think that a tool that generates potential alternative packages would be useful. As Figure 4 shows, most participants believe that the suggestions generated by our approach are useful and support the idea of having a tool to generate such suggestions. On a 5-points Likert scale, the support of such a tool has a median $= 4$ and mean $= 3.37$, where only 10% of the participants claimed it is not useful for them, and 15% indicate that it is extremely useful.

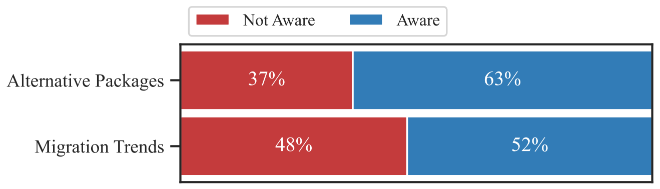

Figure 3: The awareness of participants about the suggested alternative packages and the community's migration trends.

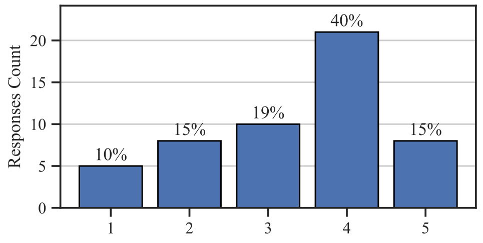

Figure 4: Survey responses to the usefulness of having a tool that generate alternative suggestions using our approach. The usefulness rated on a 5-points Likert-scale ranges from $1 =$ Not useful, to $5 =$ Extremely useful.

In the survey invitation, we provide the participants with pull request examples of dependency migrations from other projects. We ask the participants if they found the pull request examples helpful. As shown in Table VI, the majority of the participants (88%) believe that the provided examples of dependency migrations from other projects are helpful. Specifically, 79% of the participants find that the provided examples help understand the differences and the required changes in the API usage between the alternative package and the current package. Also, 75% of the participants indicate that the migration examples from other projects can help in estimating the efforts needed to migrate to the alternative

Table VI: Participants' responses on how helpful are the examples of dependency migrations from other projects?

Helpfulness	Frequency
Understanding the required API changes	79%
Estimating the dependency migration efforts	75%
Justifying the dependency migration	52%
Other	6%
Not helpful	11%

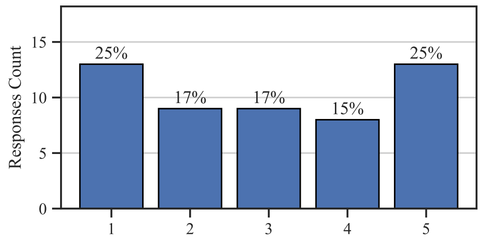

Figure 5: Survey responses on the support of migrating their current projects to use the alternative packages. The support rated on a 5-points Likert-scale ranges from 1 = Keep the current package, to 5 = Strongly advise migrating.

package in their projects. Interestingly, 52% of the participants mentioned that the explanation in the provided examples helps justify the dependency migration within their teams. Finally, only 12% of the participants did not find the examples helpful for their project. Participant P3 expressed a disagreement with the justification given in the pull request example. The participants believe that the alternative package has more dependencies, which is the opposite to what was explained in the pull request example, "Just reading briefly, but yargs has way more dependencies than commander, contrary on what is reported in the PR". Another participant mentioned that a pull request example would only be useful to them if it illustrates solving a security issue (P50: "I would never bother migrating unless there were a severe and applicable security concern ...").

Future Actions. Whether developers would act upon our suggestions is a strong indication of our approach's usefulness. We ask the participants 1) if they plan to use the alternative packages in their future projects and 2) how likely are practitioners to migrate to alternative packages in their current projects. The responses from our survey shows that 67% of the participants will use the alternative packages in their future projects. However, participants are split between supporting the dependency migrations and keeping current packages in their current project (see Figure 5). On a 5-point Likert scale, participants rate their support to migrate to the alternative package in their current projects in median = 2.

To understand why participants opt for retaining current packages, even if the package has been declining in the community, we analyze the free-text justifications from our survey participants on why they will not use the alternative package. In total, we manually examined 17 responses (out of the 52 responses) from the participants who indicated that they would not use the alternative package. Based on this analysis, we observe different reasons why they will not use the alternative package. Six participants mentioned that they would not use an alternative package as long as the current one is working. As one respondent P2 summarizes, "If it ain't broke, don't fix." While migrating to a dependency with more functionalities can be helpful, the benefits of migrating will only be achieved if the project needs the extra functionality. One of our participants (P47) expressed this concern and said "The alternative packages bundles many functions, I need just the one that the old package uses." In contrast, P3 prefers not to use a package that has more dependencies, and state "Because I value packages with little to none dependencies, one reason (of many) is the Dependabot's alerts hell, which is a huge waste of time, more often than not." Other participants indicated having less priority to migrating packages that perform a minor functionality or packages that they do not use in the production. For example, P9 said "It's just a small development time dependency used during build...as long it works - it works." Interestingly, some participants will not use the alternative packages because their current projects have low priority. For example, P13 said "The project is mostly deprecated and will be moving to a new golang based system."

Summary of RQ3: Our approach was deemed very useful for practitioners (median of 4 in 5-point Likert scale) as it showcases new alternative packages, informs about migration trends, and help estimate migration efforts. Out of our 52 participants, 67% said they consider using suggested alternatives in their future projects but were more conservative when migrating current projects due to potential maintenance risks.

VI. Discussion

In this section, we discuss our results and how our approach may help improve software maintenance activities.

The need for tools that increase community awareness. Our RQ3 results strongly suggested that even experienced JavaScript developers, that frequently search for new and better alternatives for their project dependencies, cannot keep track of ecosystem evolution. Out of the 52 respondents, almost half (48%) were not aware of migration trends, and 37% did not even know of the better alternative package. Given that, migrations are primarily motivated by better software maintainability (RQ2), there is a dire need (and opportunity) to harness complex community patterns to inform practitioners on potential maintenance problems. By recommending better alternative packages, our tool can effectively inform developers that the packages they use have become poorly maintained. Poorly maintained dependencies are a major maintenance risk for software projects, as they are unlikely to respond to bug fixing request or reported vulnerabilities [46, 47].

Our approach is scalable and would fit well in open source workflow. While our approach entails mining an entire ecosystem to identify migration patterns, the computation power needed to identify such patterns is modest. Analyzing the history of dependency changes in the entire npm ecosystem took approximately 24 hours, and new suggestions can be incrementally analyzed in minutes. Our approach can be computed centrally on a server, and be provided as a service to subscribers - similarly to Dependabot [47], informing maintainers when a new migration suggestion is identified. Given our RQ2 showed that, migrations are frequently performed in dedicated tasks, developers can validate suggestions - discard the ones not relevant for their project - and perform the desired migrations in a batch, before project releases.

Thresholds and tool configuration. Identifying useful patterns using the wisdom of the crowd requires navigating the delicate trade-off between the sensitivity and the quality of the migration suggestion. We opted to identify fewer but better quality suggestions by choosing the criteria shown in Section II-D. A much larger number of suggestions would be extracted, for example, if we relaxed the need to have the migration be performed recently, however, at the cost of recommending outdated migration suggestions. While a more in-depth analysis may identify an even better set of criteria, our criteria is a good starting point for further exploration.

VII. Related Work

Several studies proposed approaches to recommend packages to developers. Thung et al. [48] proposed an approach to recommend packages for projects based on their current dependencies, using association rule mining and collaborative filtering. Other studies targeted the same problem by using different approaches, such as multi-objective optimization [49], and pattern mining and hierarchical clustering [50]. Recently, Nguyen et al. [51] proposed a more efficient approach as it generates recommendations in a comparably less historical data. The main goal of these approaches is to tap in the missed opportunities of using available packages, based on the project's package dependencies and characteristics. However, our goal in this study is to recommend migration opportunities of better alternatives than packages already in use.

Chen et al. [52] proposed an approach for mining Stack Overflow tags to find semantically similar packages. Even though this approach can return a set of similar packages, it has no evidence of the feasibility of migrations between the alternatives [16]. Thus, researchers proposed mining historical migrations from existing software repositories, which rely on the crowd's wisdom in performing migrations to alternative packages [13, 14, 15]. However, their approaches suffer from either low recall [13, 15], or low precision [13, 14]. He et al. [16] improved the performance by utilizing multiple metrics to capture different dimensions of evidence from development histories when recommending dependency migrations extracted from other software repositories. Since this approach relies on analyzing the commits and their messages to extract migration patterns, it is sensitive to how developers divide their changes across commits and the clarity of commit messages. In contrast, our approach extracts dependency migrations based on versions without considering the individual commits, which overcomes the previous limitation. Also, our approach targets migration suggestions for packages in decline only, avoiding the problem of undesired suggestions [53].

Researchers empirically investigated dependency migrations. Kabinna et al. [54] highlight the challenges in migrating to new logging packages. Alrubaye et al. [55] analyzed several code quality metrics before and after applying dependency migrations. Others studied the role of common metrics in developer selection of packages [56, 57]. He et al. [16] proposed new four metrics (i.e., Rule Support, Message Support, Distance Support, and API Support) to rank the migration suggestions. They all use project level metrics in their approaches, while we are the first to use the centrality in the context of dependency migrations (i.e., an ecosystem level metric), which emphasizes the community interest in performing the dependency migrations.

VIII. Threats to Validity

Threats to Internal Validity. Threats to internal validity are related to experimenter bias and errors. A limitation of our approach is that it only considers dependencies between packages in the npm registry. This limitation will affect the centrality and migration patterns of packages that are not meant to be used by other packages, but other JavaScript applications, i.e., top-level packages. However, previous work has shown that using the npm registry as the sole source of changes in the dependency graph can serve as a proxy for the overall [10, 58, 59]. Future work should investigate how to incorporate JavaScript applications on the generated suggestions. Another threat concerns the process we used to filter dependency change events to our approach. To reduce noise, we opted to remove dependency change events from massive or imbalanced number dependency changes, as explained in Section II-A. This may affect our recommended patterns, as they will more likely based in projects that perform small dependency changes over time. We mitigate this effect by only considering migration patterns that reoccur across many projects. Finally, our approach may contain bugs that may have affected our results. We made our scripts and dataset publicly available to be fully transparent and have the community help verify (and improve) our approach [19].

Threats to External Validity. Our evaluation focused entirely on the npm ecosystem, which has very particular characteristics: a centralized package registry, hundreds of thousands of software packages, and a very active and popular programming language. Also, packages in the npm ecosystem are relatively small compared to modules and software components in other ecosystems and programming languages [17], which could lead to different dynamics than other ecosystems, significantly affecting the dependency migration patterns. Future work needs to investigate if a similar approach can effectively find dependency migration suggestions in other ecosystems such as PyPI and Maven.

IX. Conclusion

This paper presents an approach to extract dependency migration trends in the software ecosystem and suggests alternatives for packages that are in decline. We evaluate our approach in npm, one of the largest and most popular software ecosystems. Our evaluation showed that our approach was accurate at suggesting alternative packages (RQ1), recommended migrations that were primarily motivated by maintenance issues (RQ2), and found that developers support having a tool that utilizes our approach to suggest alternative packages (RQ3). Future work could explore avenues to improve and better operationalize our proposed approach. While we generate suggestions for one-to-one dependency migrations, it is valuable for future work to support one-to-many and many-to-many dependency migrations, where one or more packages are replacing one or more packages. Another exciting follow-up work is to propose an automated approach that uses the dependency migration examples to perform the suggested dependency migrations, refactoring the code, and ensuring the semantics are preserved during the migration.

Acknowledgement. The involvement of human participants in this study was approved by Concordia University's Research Ethics Board (protocol form number: 3000472)

References

[1] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. IEEE Press, 2017, pp. 102–112.

[2] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab, "Not all dependencies are equal: An empirical study on production dependencies in npm," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556896

[3] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "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*, ser. ESEC/FSE 2017. ACM, 2017, p. 385–395.

[4] A. Zerouali and T. Mens, "Analyzing the evolution of testing library usage in open source Java projects," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER '17, 2017, pp. 417–421.

[5] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, ser. ICSE '94. IEEE, 1994, pp. 279–287.

[6] M. Valiev, B. Vasilescu, and J. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '18. ACM, 2018, p. 644–655.

[7] GitHub, "Github sponsors," https://github.com/sponsors, 08 2022, (Accessed on 03/01/2023).

[8] O. Collective, "Raise and spend money with full transparency," https://opencollective.com/, 08 2022, (Accessed on 03/01/2023).

[9] J. Khondhu, A. Capiluppi, and K.-J. Stol, "Is it all lost? a study of inactive open source projects," in *Open Source Software: Quality Verification*, E. Petrinja, G. Succi, N. El Ioini, and A. Sillitti, Eds. Springer Berlin Heidelberg, 2013, pp. 61–79.

[10] S. Mujahid, D. E. Costa, R. Abdalkareem, E. Shihab, M. A. Saied, and B. Adams, "Toward using package centrality trend to identify packages in decline," *IEEE Transactions on Engineering Management*, vol. 69, no. 6, pp. 3618–3632, 2022.

[11] J. Coelho, M. T. Valente, L. L. Silva, and E. Shihab, "Identifying unmaintained projects in GitHub," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. ACM, 2018.

[12] J. Coelho, M. T. Valente, L. Milen, and L. L. Silva, "Is this GitHub project maintained? measuring the level of maintenance activity of open-source projects," *Information and Software Technology*, vol. 122, p. 106274, 2020.

[13] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 289–298.

[14] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014.

[15] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "Migrationminer: An automated detection tool of third-party java library migration at the method level," in *2019 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSE '19, 2019, pp. 414–417.

[16] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, ser. SANER '21, 2021, pp. 72–83.

[17] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, p. 381–416, Feb. 2019.

[18] "Stack overflow developer survey 2018," https://insights.stackoverflow.com/survey/2018/, Mar. 2018, (Accessed on 10/26/2018).

[19] S. Mujahid, D. E. Costa, R. Abdalkareem, and E. Shihab, "Replication package: Where to go now? finding alternatives for declining packages in the npm ecosystem," Oct. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5548231

[20] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the past – analysing backporting practices in package dependency networks," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[21] F. Inc., "react on the npm website," https://www.npmjs.com/package/react, Mar. 2121, (Accessed on 09/12/2021).

[22] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 107 – 117, 1998, proceedings of the Seventh International World Wide Web Conference.

[23] GitHub, "Api documentation of graph ql," https://docs.github.com/pt/graphql, 02 2023, (Accessed on 03/03/2023).

[24] G. Gousios and A. Zaidman, "A dataset for pull-based development research," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 368–371.

[25] J. P. Manager, "npm," https://www.npmjs.com/, 01 2023, (Accessed on 05/01/2023).

[26] M. McHugh, "Interrater reliability: The kappa statistic," *Biochemia medica*, vol. 22, pp. 276–82, 10 2012.

[27] J. L. Fleiss and J. Cohen, "The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability," *Educational and Psychological Measurement*, vol. 33, pp. 613–619, 1973.

[28] T. Inc., "Ember.js - a framework for ambitious web developers," https://emberjs.com/, (Accessed on 05/01/2023).

[29] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[30] J. Cohen, "A coefficient of agreement for nominal scale," *Educational and Psychological Measurement*, vol. 20, pp. 37–46, 1960.

[31] S. Silbermann, "Switch to cross-fetch," https://github.com/mui-org/material-ui/pull/19644, Feb. 2020, (Accessed on 09/07/2021).

[32] N. Hardcastle, "Replace raven with sentry-sdk," https://github.com/OpenNeuroOrg/openneuro/pull/1040, Jan. 2019, (Accessed on 09/07/2021).

[33] P. Parsa, "Use terser-webpack-plugin," https://github.com/nuxt/nuxt.js/pull/3928, Sep. 2018, (Accessed on 09/07/2021).

[34] O. Pakers, "Replace bcrypt with bcryptjs," https://github.com/keystonejs/keystone/pull/2053, Dec. 2019, (Accessed on 09/06/2021).

[35] B. Waterloo, "Switch vscode-azureappservice to dayjs," https://github.com/Microsoft/vscode-azuretools/pull/808, Oct. 2020, (Accessed on 09/07/2021).

[36] A. Slagle, "Switched to dompurify for html sanitization," https://github.com/NYPL-Simplified/opds-web-client/pull/115, Apr. 2016, (Accessed on 09/07/2021).

[37] J. Bernhardt, "Update packages," https://github.com/compose-us/todastic/pull/19, Jul. 2019, (Accessed on 09/09/2021).

[38] L. Quixada, "Dropped dependency isomorphic-fetch in favor of cross-fetch (react native compatible)," https://github.com/apollographql/apollo-fetch/pull/71, Oct. 2017, (Accessed on 08/31/2021).

[39] D. A. Dillman, *Mail and Internet surveys: The tailored design method–2007 Update with new Internet, visual, and mixed-mode guide*. John Wiley & Sons, 2011.

[40] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the impact of using trivial packages: An empirical case study on npm and PyPI," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1168–1204, 2020.

[41] Y. Golubev, M. Eliseeva, N. Povarov, and T. Bryksin, "A study of potential code borrowing and license violations in java projects on github," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 54–64.

[42] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '16, 2016, pp. 334–344.

[43] GitHub, "Github search api," https://docs.github.com/pt/rest/search?apiVersion=2022-11-28, 11 2022, (Accessed on 03/12/2022).

[44] R. P. L. Buse and T. Zimmermann, "Information needs for software development analytics," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 987–996.

[45] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann, "Improving developer participation rates in surveys," in *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '13. IEEE, 2013, pp. 89–92.

[46] S. Zajdel, D. E. Costa, and H. Mili, "Open source software: An approach to controlling usage and risk in application ecosystems," in *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 154–163. [Online]. Available: https://doi.org/10.1145/3546932.3547000

[47] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 254–265.

[48] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *2013 20th Working Conference on Reverse Engineering*, ser. WCRE '13, 2013, pp. 182–191.

[49] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[50] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," *Journal of Systems and Software*, vol. 145, pp. 164–179, 2018.

[51] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "Crossrec: Supporting software developers by recommending third-party libraries," *Journal of Systems and Software*, vol. 161, p. 110460, 2020.

[52] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q amp;a discussions – incorporating relational and categorical knowledge into word embedding," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16, vol. 1, 2016, pp. 338–348.

[53] L. Erlenhov, F. G. d. O. Neto, and P. Leitner, "An empirical study of bots in software development: Characteristics and challenges from a practitioner's perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 445–455.

[54] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the apache software foundation projects," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 154–164.

[55] H. Alrubaye, D. Alshoaibi, E. Alomar, M. W. Mkaouer, and A. Ouni, "How does library migration impact software quality and comprehension? an empirical study," in *Reuse in Emerging Software Engineering Practices*, ser. ICSR '20, S. Ben Sassi, S. Ducasse, and H. Mili, Eds. Cham: Springer International Publishing, 2020, pp. 245–260.

[56] F. L. de la Mora and S. Nadi, "An empirical study of metric-based comparisons of software libraries," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 22–31.

[57] S. Mujahid, R. Abdalkareem, and E. Shihab, "What are the characteristics of highly-selected packages? a case study on the npm ecosystem," *Journal of Systems and Software*, vol. 198, p. 111588, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222002643

[58] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "Deprecation of packages and releases in software ecosystems: A case study on npm," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[59] ——, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.