Towards Using Package Centrality Trend to Identify Packages in Decline

Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams

***Abstract*—Due to their increasing complexity, today's software systems are frequently built by leveraging reusable code in the form of libraries and packages. Software ecosystems (e.g., npm) are the primary enablers of this code reuse, providing developers with a platform to share their own and use others' code. These ecosystems evolve rapidly: developers add new packages every day to solve new problems or provide alternative solutions, causing obsolete packages to decline in their importance to the community. Developers should avoid depending on packages in decline, as these packages are reused less over time and may become less frequently maintained. However, current popularity metrics (e.g., Stars, and Downloads) are not fit to provide this information to developers because their semantics do not aptly capture shifts in the community interest.**

In this paper, we propose a scalable approach that uses the package's centrality in the ecosystem to identify packages in decline. We evaluate our approach with the npm ecosystem and show that the trends of centrality over time can correctly distinguish packages in decline with an ROC-AUC of 0.9. The approach can capture 87% of the packages in decline, on average 18 months before the trend is shown in currently used package popularity metrics. We implement this approach in a tool that can be used to augment the *npms* metrics and help developers avoid packages in decline when reusing packages from npm.

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

I. Introduction

SOFTWARE ecosystems have changed the way we develop software. Platforms like npm, PyPI, and Maven enable developers to easily reuse code from other projects in the form of packages [1], boosting development productivity [2], and improving software quality [3]. As such, these ecosystems are becoming extremely popular and large. For example, the node package manager (npm) alone has more than 1.4 million packages to date and is seeing exponential growth [4].

The large size and rapid evolution of these ecosystems has its downsides as well. For example, new (and often better) packages are continuously being introduced [5, 6, 7, 8], making other, once popular packages, obsolete, dormant or even deprecated [9]. As such, it is becoming increasingly important for application developers to ensure that they choose the right packages from the ecosystem.

Although prior work examined projects that are unmaintained [10, 11], to the best of our knowledge, little attention has focused on identifying packages that lose popularity over time (i.e., are in decline). At the same time, current popularity metrics that are commonly used by developers to select packages, such as downloads and stars, are not adequate to capture a shift in community interest. For example, the number of downloads represents not only the number of times a package is installed on its own, but also the number of times it is installed as a dependency of other packages. Hence, the popularity of a dependent package could heavily impact the number of downloads of its dependencies [12]. Also, the number of stars a package is linked to its repository, which may include many other packages and is unlikely to decrease to reflect interest shift over time [13, 14].

Therefore, in this paper we use the package's centrality as a proxy of community interest. Community interest drives packages to evolve, 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 GitHub Sponsors,[1] Open Collective,[2] and Tidelift.[3] On the other hand, packages that are declining in community interest are reused less over time, may become less actively maintained, and in more extreme cases, even become abandoned [9, 15]. Furthermore, the decline in community interest of a package may indicate that a better solution is drawing attention in the ecosystem, and developers are migrating to a package that better suits their needs.

Hence, our aim is to effectively identify packages that may be in decline. To do so, we use the package centrality to identify declining community interest. By definition, centrality is a measure of the prominence or importance of a node in a social network [16]. Centrality has been used in many fields e.g., in finance to measure the stability of banks in financial networks [17], in electrical engineering to rank the importance of components in network infrastructures [18, 19], and other fields including computer science and software engineering [20, 21]. In our context, centrality allows us to rank the packages based on the popularity/importance of packages that depend on them. Specifically, we use the PageRank algorithm to evaluate the shift in their centrality over time.

S. Mujahid, D. E. Costa, and E. Shihab are with the Data-driven Analysis of Software (DAS) Lab at the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. Email: {suhaib.mujahid, diego.costa, emad.shihab}@concordia.ca

R. Abdalkareen is with the School of Computer Science, Carleton University, Ottawa, Canada. Email: rabe.abdalkareem@carleton.ca

M.A. Saied is with the Department of Computer Science and Software Engineering, Laval University, Quebec City, Canada. Email: mohamed-aymen.saied@ift.ulaval.ca

B. Adams is with the School of Computing, Queen's University, Kingston, Canada. Email: bram.adams@queensu.ca

Manuscript accepted October 18, 2021.

[1] https://github.com/sponsors

[2] https://opencollective.com

[3] https://tidelift.com

The intuition is that packages that have a consistent decrease in the centrality ranking are likely to be packages in decline. Hence, package developers should be careful when depending on such packages.

We evaluate our approach on the npm ecosystem. We do so since JavaScript is one of the most popular programming languages [22] and npm is the largest growing ecosystem to date [4]. The popularity and scale of the npm ecosystem makes it an ideal candidate for our study. We evaluate the effectiveness of using package centrality in identifying npm packages that are in in decline. We formalize our study through the following research questions:

- RQ1: How effective is our approach in detecting packages that are in decline?
- RQ2: How early can our approach detect packages that are in decline?
- RQ3: How does our approach compare to other metrics in detecting packages that are in decline?

Our findings show that our approach can detect 87% of packages in decline with high accuracy, on average 18 months before current popularity metrics show the decline. Also, we find that our approach can detect packages in decline more than 16 months (on average) before such packages are deprecated. Lastly, we find that our approach complements commonly used popularity metrics such as dependents, downloads, stars, and forks when detecting packages in decline.

Our work makes the following contributions:

- Propose a scalable approach to detect packages in decline using package centrality.
- Empirically evaluate our approach on the npm ecosystem.
- Create a tool prototype to facilitate the usage of our approach by practitioners.
- Make all of our dataset (i.e., the collected data, analysis results, scripts) publicly available to support replication and future research [23].

Paper organization. The remainder of this paper is organized as follows: We motivate our work in Section II with an example of a popular package in decline. Section III details our approach, from data collection to computing centrality trends to find packages in decline. In Section IV we explain how we collect and curate the baselines we use to evaluate our approach. Section V presents the findings of our empirical study by answering our three research questions. We present a tool prototype to utilize our approach in Section VI. Section VII discusses the related work and Section VIII describes the threats to validity. Finally, we conclude in Section IX.

II. Motivating Example

To illustrate the idea of using package centrality in determining a shift in community interest, we present the example of the Moment.js package. Moment.js is a JavaScript library for parsing, validating, manipulating, and formatting dates. This is a highly-used package, used in more than 1 million websites, including major companies[4] such as CNN, Microsoft Teams, LinkedIn and Dropbox. Moment.js was developed using a now old-fashioned JavaScript packaging method, including all its functionalities in a single bloated JavaScript class. Consequently, all websites that use Moment.js have to include the entire package regardless of the feature used, which incurs in an unnecessary overhead for website applications [24].

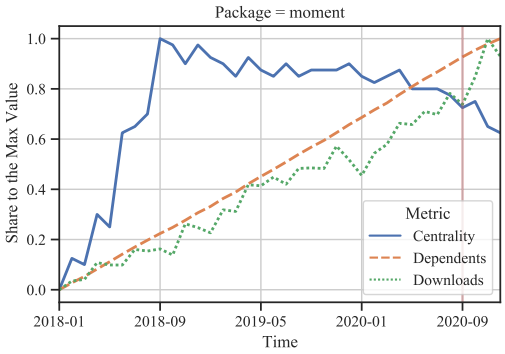


Figure 1: Evolution of the Moment.js package on the centrality (PageRank), the number of downloads, and the number of dependents. The red vertical line indicates the time where maintainers reported Moment.js is now a legacy project. We normalize metric values using the min-max method where values range from 0 and 1 [28].

"Moment was built for the previous era of the JavaScript ecosystem. The modern web looks much different these days."

Since 2018, alternatives to Moment.js (e.g. date-fns and Day.js) have become more and more popular by providing similar functionality without incurring the overhead that Moment.js incurs. Hence, the npm community started shifting towards using more lightweight packages. This shift includes migrating well-established open source projects like Google Chrome's Lighthouse, Vault by HashiCorp, and Web Stories by Google from Moment.js to other alternative packages [25, 26, 27].

The popularity of the alternative packages led to a consistent decrease in Moment.js's centrality in the ecosystem starting in September 2018, which can be seen clearly in Figure 1. On September 15th, 2020, the maintainers of Moment.js issued a statement in the README file indicating that the package is now a legacy project. While maintainers have committed to still maintain the project, they recommend that users choose a different package.

The community's shift from using one of npm's most used packages to other alternatives was public knowledge; however, none of the common popularity metrics, including the metrics used by the npm search engine (*npms*) were able to capture this phenomenon. In fact, the number of downloads of Moment.js continued to increase (as shown in Figure 1) as well as the number dependent packages. As of January 2021, the npm registry shows that 49,544 npm packages depend on Moment.js and it is downloaded more than 16 million times a week. The only metric that showed Moment.js's important decrease in npm was centrality, which started to decrease as

[4]Reported by wappalyzer.com in January 2021

early as October 2018, the same year that alternative packages started to become more popular.

There are a couple of possible reasons why the number of downloads and dependents did not capture the decline of Moment.js. First, since thousands of projects already use Moment.js, it will continue to be downloaded every time any of these projects get installed. Even when these projects migrate to use alternative packages, it will take much longer to reflect on the number of downloads due to technical lag where developers take a long time to update their dependencies [29]. Second, as npm continues its exponential growth, newly created packages may still depend on Moment.js and substitute the core community that has migrated to the alternative solutions. Package centrality, calculated with PageRank, accounts for not only the sheer number of dependents, but the importance of dependents in the network, which aptly captures the decline of Moment.js. This example motivated us to investigate if package centrality trends can be used to identify packages that have declined in the community interest.

III. Approach

In this section, we explain our approach that uses the trend in the package's centrality in the npm ecosystem and detect packages in decline.

A. Calculating Centrality Trends

Since the core idea of our approach is to use centrality, we need to efficiently calculate the centrality trends of packages. We first build a dependency graph containing all packages in npm as nodes, and their dependency relationships as edges. We update this graph monthly with newly established dependencies and packages and compute the centrality metric for all packages. Each month, we rank the packages based on the value of their centrality metric. In the following, we explain the attributes of our dependency graph, then, we describe our approach, illustrated in Figure 2, which includes how we: i) collect and format the required metadata to build the dependency graph incrementally, and ii) build the dependency graph each month to compute the centrality metric for all packages in the npm ecosystem.

Attributes of Our Dependency Graph. In order to use the package centrality as an indicator for packages in decline, our dependency graph needs to have two important properties:

1) **Version insensitive nodes:** the nodes in our graph represent npm packages, regardless of their versions. For instance, the popular package React has 298 distinct versions released in the npm registry, but we represent it by only one node in our dependency graph. We do this because we are interested in capturing the usage shift without being affected by the technical lag in the dependency network, caused by developers taking a long time to update a dependency version [29, 30].
2) **Release sensitive edges:** an edge A $\rightarrow$ B in our graph represents the dependency between the latest released version of package A on any version of package B. Once a new release of package A no longer depends on B, our dependency graph needs to reflect that by removing the A $\rightarrow$ B edge. However, we do not consider backport versions as the latest released versions since they are not consistent with the package evolution time series.

To better illustrate how this dependency graph is built, Figure 3 presents an example of one package's dependencies and how they are reflected in our dependency graph. As shown in Figure 3, the graph in each month (January and February) uses the latest version of Package A to add the edges from node A to its dependencies, but disregards the versions of the dependencies (packages B and C). Once package B is removed from A's dependencies (in February), we remove the edge A $\rightarrow$ B in the dependency graph. It is important to note that, by not accounting for versions in the nodes, this dependency graph is different from the dependency graph that npm resolves to install new package versions when running the `npm install` command [31].

Extracting Dependency Change Events. To build the npm dependency graph, we need to extract and process all events that changed dependencies for all npm packages. In our study, we need to process two types of dependency change events for all npm packages: 1) the addition of a new package dependency and 2) the removal a package dependency. Since our dependency graph does not consider the package versions in their nodes, there is no need to account for events updating a package dependency version. We use the npm registry database to extract all the package dependency change events. The npm registry keeps a copy of the `package.json` file of all npm packages in its database for all package versions. The `package.json` file includes the list of maintainers, package description, keyword, license, the address of the source code repository, and the list of package dependencies. The registry stores each package as a document that contains its metadata.

The npm registry is powered by an Apache CouchDB database, which has a feature to set up a continuous stream of its data [32]. The feature is typically used to set up continuous replication from the registry database. We utilize this feature to retrieve a stream of all documents from the npm registry (Step ①). For each document in the stream, we filter out the irrelevant documents (e.g., design documents) and for each package we collected the `package.json` file for each of its versions.

When we build the monthly dependency graph, we only use the most recent version of each package version to create our dependency graph. Hence we order every package release by its release date. However, not all releases represent the stage of the package project at the target time. Backports are commonly employed by package maintainers to fix older releases, and they could include old dependencies that no longer appear in the package's latest releases. Hence, we filter out any release with a lower semantic version than its predecessor in relation to their respective release date (Step ②). For example, package A has released the version `3.6.0` in March 2020, but released a backport fix `2.1.0` in April 2020. Because the version `2.1.0` is smaller than version `3.6.0`, we disregard the version `2.1.0` in our analysis.

Finally, we extract the changes in the list of dependencies

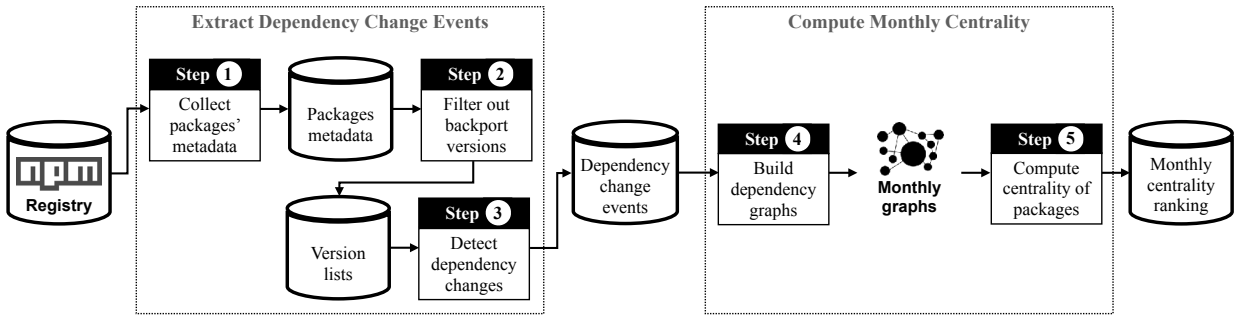


Figure 2: The approach to calculate centrality trends.

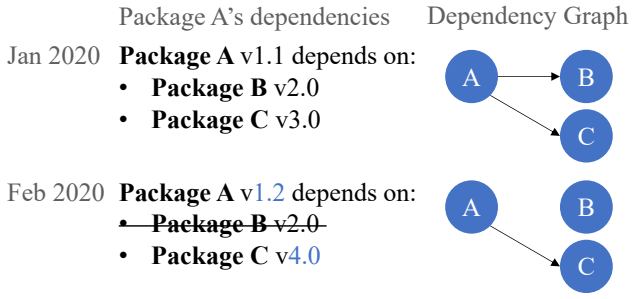


Figure 3: Illustration of our dependency graph build process

between versions (Step ③). We represent each change in the dependencies list as a dependency change event, which can be either an add or a remove event. When a package releases its first version, we consider each of the dependencies required by that version as an add dependency change event. In the following versions, we compare the list of dependencies on each version with the list in the previous version. If the dependency is absent in the newer version, we consider it to be a remove event; conversely, if the dependency is absent in the older version, we consider it an add event.

Computing Monthly Centrality. To obtain monthly snapshots of the centrality trends, we compute the centrality for the packages in the npm ecosystem each month. Consequently, we need to build a dependency graph at each month of analysis. Building separate graphs from scratch for every month can be an expensive operation and unpractical option, particularly for npm, which contains more than a million packages. To address this, we build the dependency graphs incrementally using the add and remove dependency change events that we explained previously.

In this study, we are interested in investigating the package centrality trends since the creation of the npm ecosystem. In particular, we study the period from December 2010 to December 2020. To do so, we build the first graph up to December 2010 and calculate the centrality for every package in that graph (Step ④). Then, for each month, we update the graph snapshot to reflect the monthly changes in the ecosystem. In total, we build 121 different versions of the dependency graph for the npm ecosystem, one for each month between December 2010 and December 2020.

We use the monthly dependency graphs to compute the centrality of packages in the npm ecosystem (Step ⑤). In order to compute the centrality, we use the Google PageRank algorithm [33, 34]. The algorithm is commonly used to rank software artifacts, e.g., JavaScript packages [6, 35] and Java components [36]. The PageRank algorithm is a variant of the Eigenvector Centrality metric, which measures the importance of each node within the graph based on the number of incoming edges and the importance of the corresponding source nodes. The underlying assumption of PageRank is that a node is only as important as the nodes that link to it [37, 38]. In our study and through the use of PageRank, the package centrality score is affected by both the number of dependent packages and the score of the dependent packages themselves. Thus, packages obtain higher scores if their dependent packages themselves have high scores.

However, the centrality value of nodes in PageRank decays over time as the network grows [39]. This may impact the evolution analysis and means it is not meaningful to compare centrality values of packages on different periods as these will always tend to decrease (at least for growing networks). To address this, we focus instead on analyzing the ranking of the nodes' centrality. Once we compute the centrality for all packages on a particular month, we rank the packages based on their centrality values $(v_1, v_2, ..., v_n)$ where v_1 is the most central package and v_n is the least central package similar to prior work [6]. Finally, we invert the ranking in negative values $(-1 \times n)$ to give a higher ranking value to the more central packages, and make the centrality ranking comparable to other metrics (e.g. downloads), where a higher value means higher importance. With this, we have the centrality ranking position evolution for each package in the npm ecosystem since its creation up until December 2020.

B. Detecting Packages In Decline

Now that we have the evolution of all packages' centrality rankings, we use it to provide a reliable method to identify packages in decline. To classify a package as in decline, we use its centrality trend of the latest six months. We fit a linear function using the least-squares regression [40], then we analyze the slope (m) of the trend to identify a package in decline. In our study, a package is classified in decline if its centrality trend shows a significant negative slope:

1) **Slope:** the slope of the centrality trend for the last six months should be $m < v$, with default $v = 0$.

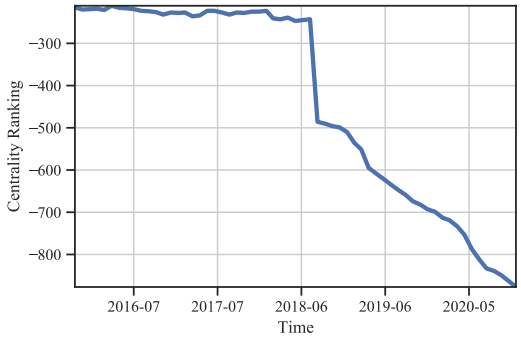


Figure 4: Example of a package trend in decline.

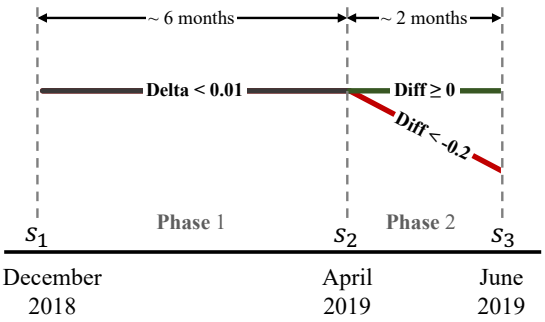


Figure 5: Timeline used to select validation baseline from *npms*.

2) **P-Value:** to test whether the negative slope is statistically significant, we perform the Wald Test with a conservative p-value (p) threshold, i.e., $p < \alpha$, with default $\alpha = 0.001$ [41]. The Wald Test is a way of testing the significance of particular explanatory variables in a statistical model.

In practice, our approach classifies packages as in decline when they have consistently fallen down in the npm centrality rankings for six months. Figure 4 shows an example of the package istanbul-api, which is classified as in decline, with a clear decrease in the centrality rankings starting from mid 2018. This decline can be justified by the incompatibility of the package with new Javascript features [42], which led to the deprecation of the package later in April 2019 [43].

IV. Evaluation Datasets

To obtain a baseline for our approach, we devise a dataset containing packages in decline and packages not in decline, so we can evaluate if our approach can reliably report packages in decline. Unfortunately, there is no existing large dataset that captures the shifts in community interest we aim to evaluate.

To compensate for the absence of this ideal dataset, we build three different baseline datasets. First, we build a corpus using metrics from the official search engine of npm (*npms*) to evaluate if centrality can detect packages in decline before *npms* (Section IV-A). Second, we collect data from the largest survey of the JavaScript community conducted by Benitte and Greif [44], which asked the opinion of more than 20 thousand developers about 20-30 popular npm packages (Section IV-B). With this baseline we aim to evaluate if centrality can capture the satisfaction/dissatisfaction of developers using the trend in centrality right before the survey took place. Third, we craft a dataset of deprecated packages to evaluate if our approach can help identify the decline in popularity well before the maintainers deprecated the packages (Section IV-C).

A. Extracting npms Validation Baseline Corpora

One of the most reliable platforms developers use to select npm packages for their projects is the official npm search engine, *npms* [45, 46]. The *npms* engine continuously analyzes the npm ecosystem, and collects 27 package metrics from different sources (e.g. package repositories on GitHub). Using the collected metrics, a final score for each package is calculated based on three different aspects i.e., quality, popularity, and maintenance [46, 45]. The higher the score of a package, the more popular, better quality and better maintained the community perceives the package to be. Hence, a steep decline in a package score can be used as an indicator of a package in decline and a stable or increasing score can be an indicator of a package not in decline.

It is important to note however that we want to evaluate the hypothesis that *centrality can better identify shifts in community interest than currently used metrics*. Hence, we want to craft a dataset that allows us to use *npms* score as validation, but is not directly influenced by the *npms* score. To this aim, we craft the dataset using a multi-phase approach, as illustrated in Figure 5. We first select packages that have shown a stable *npms* score during a period (Phase 1), and use this same period to evaluate the centrality of a package. Because the score metric is stable during this period, one would not be able to classify the packages in decline from not in decline just by analyzing *npms* score, and we can be sure centrality is not influenced by already reported metrics. Then, in the subsequent period (Phase 2), we label packages in decline as the ones that have experienced a sharp decline in the *npms* score and label packages not in decline as the ones that have either remained stable or increased their *npms* score. Using this process to craft the baseline, we also have a starting date for packages in decline given by the *npms* score, which is at the earliest the start of Period 2. We can use this point in time, to evaluate how much in advance (if any) our approach can detect that a package is in decline before the decline is shown in the *npms* score.

One limitation of using the *npms* metrics is that *npms* does not store the historical values of its packages' score. We cannot pick any period interval for Phase 1 and Phase 2 and are limited by the snapshots of the entire *npms* score ranking we collected in the past. We collected *npms* packages' scores on December 2018, April 2019, and June 2019 and we use the period of December 2018 to April 2019 as Phase 1, and use the period of April 2019 to June 2019 as Phase 2 of our dataset baseline.

All *npms* scores vary from 0 (very low) to 1 (perfect score). We start crafting our dataset by selecting packages that have a score 0.7 or higher, to prevent our analysis from focusing on very low-quality packages. As we showed in Figure 5, in Phase

1 we consider all packages that have a score variation smaller than 0.01, which indicates to be relatively stable. Then, we label as packages in decline, all packages that have exhibited a negative change in the *npms* score between S_2 and S_3 by more than 0.2 score points. We label as packages not in decline, all packages that have exhibited the same score or higher between S_2 and S_3. At the end of this process, this dataset contains a total of 4,457 packages, with 2,259 being labeled as in decline and 2,198 labeled as not in decline.

The three thresholds used in the above methodology were determined as follows: the first threshold of 0.01 is the tolerance in the *npms* score deltas in Phase 1. This threshold equals the mean value of changes in the *npms* score between S_2 and S_3 and it is small enough to guarantee that package scores are stable for at least 6 months before April 2019. The second threshold 0.2 is the minimum decrease in the *npms* score to label a package as in decline. This threshold is equal to the value of standard deviation over the *npms* scores and it is large enough to capture the significant score changes. The last threshold, 0.7, is the minimum *npms* score for a package on S_3 to be considered in our baseline dataset. This will minimize the risk of mislabeling our baseline by including low-quality packages with very low npms scores and it is a good compromise between the dataset size and quality.

B. Survey Validation Baseline Corpus

We want to evaluate if our approach can capture the shifts on the interest and satisfaction of the `npm` community with popular packages. While we cannot craft a dataset that reliably captures the `npm` community interest without surveying a very large sample of JavaScript developers, we opted to use the data from the largest survey available on the JavaScript ecosystem: the State of JavaScript survey [44].

The State of JavaScript survey is an extensive survey conducted by Benitte and Greif [44] to assess the JavaScript community's views. In 2019, the survey had a total of 21,717 respondents all across the globe [47]. The survey's primary focus is to ask JavaScript developers their opinion on a set of popular `npm` packages. Then, the survey ranks each package according to four categories:

1) **Awareness:** share of total respondents that reported to have heard about the package. This category includes both developers who have experience using the package and developers never use the package before.
2) **Usage:** share of total respondents that have used the package in their projects. This category does not consider if the developer is satisfied with using the package.
3) **Interest:** share of respondents who did not use the package but are interested in using it in the future.
4) **Satisfaction:** share of respondents that have used the package in the past and will continue to use it.

To use the survey results, we use its GraphQL API[5] to retrieve the summary of the responses for each package.

[5]https://graphiql.stateofjs.com

C. Deprecated Packages Corpus

With this third corpus, we want to evaluate if our approach can help identify packages in decline that have eventually been deprecated by maintainers. Deprecated packages should not be reused by other packages or JavaScript applications and `npm` warns developers when they install deprecated packages. The goal of our analysis is to evaluate if centrality trends can capture the decline in the community interest well before the package is flagged as deprecated, which can help developers to migrate from these packages while they are still being maintained.

To craft this dataset, we need to collect a list of deprecated packages from the `npm` ecosystem. Similar to Section III, we started by retrieving the metadata for all packages from the `npm` registry. Then we capture metadata for packages with a deprecation message, which left us with a list of 44,857 packages. However, developers use the `npm` deprecation feature for various reasons, including renaming or merging packages. The following quote is an example of a deprecation message for a package whose maintainers used the depreciation feature to change the package name.

"Jade has been renamed to pug, please install the latest version of pug instead of jade [48]."

To create a valid list of deprecated packages, we select the top 1,000 deprecated packages based on their *npms* score on June 16th, 2019. Then we manually classify packages to filter out cases where they are not an actual deprecation. For this aim, first, we verify if the deprecation note discloses clearly that a package is actually deprecated. If the deprecation message is not clear, we check the project status from the package's readme file, then the repository's readme file. If needed, we follow relevant links in the deprecation messages or the readme files to remove ambiguity. Finally, if the deprecation message mentions another package's name, we check if both are pointing to the same repository; if so, we examine the repository and its history to classify the case as a rename or not. After applying our manual classification process, we find that only 556 out of the 1,000 packages are actual package deprecation cases. We use these 556 packages in our analysis later in the study.

V. Results

This section describes our research questions. For each research question, we explain its motivation, illustrate our approach to answer the question, and discuss the findings.

RQ1: How effective is our approach in detecting packages that are in decline?

Motivation. In this question, we investigate the performance of our approach of using the centrality trend to identify packages in decline. The decline of package centrality could be a symptom that better alternatives have emerged or a shift happened in the community interest. In the scientific literature, centrality has been used in many disciplines such as social networks to identify the central node of a network

(e.g., [18, 20, 19, 21]) and software engineering to understand the significance of software components (e.g., [6, 36]). If the approach can aptly capture packages in decline, it can be embedded in package search engines, such as *npms*, to increase developers' awareness of the community interest and help them make a better-informed decision to select or reevaluate their package dependencies.

Approach. We craft a baseline as described in Section IV-A to evaluate our approach as a binary classification problem. Then we use our approach to classify packages into two classes: in decline and not in decline.

As mentioned in Section IV-A, packages labeled in decline are packages that have experienced a sharp decline in the *npms* ranking in a short period of two months, i.e., between S_2 and S_3. We calculate the centrality in the last six months before S_2, when the packages were still stable in the *npms* rankings. This ensures that we evaluate if the centrality can be used as an early detector of packages in decline that only later will be observed in the *npms* rankings. Then, as described in Section III-B, we classify packages that have a negative centrality trend slope (i.e., $m < v$ with default $v = 0$) as in decline and other packages as not in decline.

To evaluate the performance of our approach in identifying packages in decline, we report the well-know performance measures: precision (P), recall (R), and F_1 score. In the context of our evaluation, precision is the percentage of packages classified as in decline that are actually in decline (i.e., $Precision = \frac{T_p}{T_p+F_p}$), where T_p is the number of packages labeled as in decline that are correctly classified as in decline; F_p denotes the number of not in decline packages classified as in decline. Recall is the percentage of packages that correctly classified as in decline relative to all of the packages that are labeled as in decline (i.e., $Recall = \frac{T_p}{T_p+F_n}$), where F_n measure the number of packages in decline that classified as not in decline. We then combine both precision and recall using the well-known F_1 score (i.e., $F_1 = 2 \times \frac{P \times R}{P+R}$).

In addition, to mitigate the limitation of choosing a fixed slope threshold (i.e., $v = 0$) when calculating precision and recall, we also present the Area Under the Receiver Operating Characteristic Curve (ROC-AUC) value. ROC-AUC is computed by measuring the area under the curve that plots the T_p rate against the F_p rate while varying the slope threshold used to determine if the approach should classify a package as in decline or not. The ROC-AUC's main merit is that it reports the performance independently from the used threshold; it is also robust toward imbalanced data since its value is obtained by varying the classification threshold over all possible values [49, 50]. The ROC-AUC has a value that ranges between 0 and 1, where a higher ROC-AUC value indicates better classification performance.

Results. As shown in Table I, we evaluate our approach on 4,457 `npm` packages where 2,259 are labeled as in decline and 2,198 are labeled as not in decline. The results show that our approach of using the centrality trends correctly identifies 87% of the packages in decline with a precision equal to 0.80. That is, for every five packages classified as in decline, four were correctly classified and one was wrongly flagged as in decline. This indicates that our approach can aptly identify packages in decline before they are actually shown in the *npms* rankings, with an F_1 score of 0.83 and ROC-AUC of 0.90 As shown in Figure 6. The figure shows the false positive rate on the x-axis and the true positive rate on the y-axis, while the solid line represents the value of each of them based on a range of possible thresholds.

Table I: Results of using the centrality trend to classify packages from the *npms* validation baseline.

Dataset	Total cases	4,457
	In decline	2,259
	Not in decline	2,198
Performance	True Positive (T_p)	1,969
	False Positive (F_p)	498
	Precision (P)	0.80
	Recall (R)	0.87
	F_1 score	0.83
	ROC-AUC	0.90

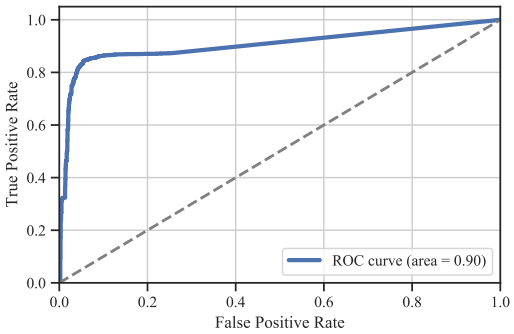


Figure 6: ROC curve with the AUC value for the evaluation based on the *npms* baseline.

We analyze the 290 packages that were in decline, but where our approach could not identify their decline using centrality. Out of the 290 cases, 217 (74.83%) packages exhibited a centrality decrease only after April 2019 (S_2), showing that in these cases the *npms* metrics decrease before the centrality. Figure 7 shows examples of packages that our approach could not detect packages in decline in advance of *npms*. In the figure, both packages show a decrease in centrality before April 2019 (S_2). However, our approach requires a statistically significant decrease over a six months period, with a very conservative default threshold $\alpha = 0.001$ to detect the packages as in decline. Hence, our approach detected the packages as in decline after S_2, when the decline became statistically significant.

We also examine the 498 packages that were not in decline, but where our approach wrongly identifies them as in decline. We observe that out of the 498 cases, 384 (77.11%) packages have less than 100 dependent packages. In the rest of the false positive cases, we observe that most of the packages (112 out of 114) have their number of dependents increasing, however their centrality is decreasing. For example, the package mongoose is a popular object modeling tool whose dependents increased from 4,995 to 5,568, whereas its centrality ranking declined from -443 to -484. The main factors behind these

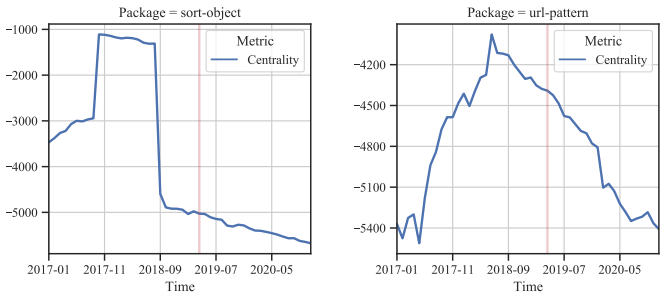


Figure 7: Examples of packages that our approach only detected the decline after the *npms* score. The red vertical line indicates the time of S_2.

false positive cases can be explained by the following:

1) The dynamic of the centrality ranking tends to punish packages that do not gain more dependents (directly or indirectly) on them compared to other packages in the same ranking tier. In the mongoose example, even with the 11.47% increase in the number of dependents, the number of dependents and their centrality were not enough to maintain the centrality ranking compared to other packages in the same ranking tier.
2) In packages with a small number of dependents, the centrality trend can be affected by a small number of community members that do not reflect the overall community interest. This could explain 52 (17.93%) of the 498 false negative cases and 384 (77.11%) of the 498 false positive cases.

Impact of Moving Averages. Simple moving averages (SMA) is a technique used to reduce the noise in the time-series data [51]. In this RQ, we use the trend of the monthly centrality rankings to detect packages in decline. However, using the SMA to smoothen the trends may result in improving the performance of our approach. In our context, we experiment using the technique to reduce the effect of noise in the monthly centrality data. To do so, we re-run our experiments on the *npms* validation baseline. For each package in the baseline, we compute the simple moving averages (based on 4 months average) for its monthly centrality rankings. Next, we apply our approach in detecting the centrality decline on the SMA values. The result of the experiment shows that incorporating the moving averages improved the precision of our approach from 0.80 to 0.85. However, it slightly decreases our approach's recall from 0.87 to 0.83, while keeping the F1-score almost constant (from 0.83 to 0.84). Finally, since using the moving averages requires more extended history, the number of packages that our approach can be applied on is reduced slightly from 4,457 to 4,272 packages.

The result shows that our approach can correctly detect **87%** of packages in decline with a precision of 0.80, an F1-score of 0.83 and an ROC-AUC of 0.90.

RQ2: How early can our approach detect packages that are in decline?

Motivation. Once we learned that our approach is effective in identifying packages in decline, we would like to know how early in advance can our approach detect the decline. Identifying packages in decline as early as possible is essential for taking proactive action to mitigate the decline of the package. Also, it increases the awareness of the community about possible better alternatives by allowing developers to avoid selecting declining packages and to pay more attention to the alternatives that are increasing in centrality. Package maintainers can also use our approach as a sign of a decrease in community interest in their package, which can motivate them to remediate possible causes of dissatisfaction or make them focus on other solutions altogether. Furthermore, developers that reuse packages can use the centrality trend as an early indicator of decline to look for alternatives long before their dependencies become unmaintained.

Approach. To evaluate how early our approach can detect packages in decline, we employ a sliding window technique. Since we calculate centrality at the granularity of months, we slide the analysis window back in time, sliding our window of six months one month at a time. We recalculate the in decline analysis after each window sliding (i.e., month) by applying the same method explained in Section III-B. We continue this process as long as the in decline analysis continues to identify the package as in decline.

Note that since we use a 6-month window to detect the deceline, when we report that our approach captured a package in decline 4 months in advance, this means that the slope of the centrality trend consistently decreased in the 6 months prior to these 4 months. That is, the package is exhibiting a decrease in the centrality rankings for up to 10 months.

We used our three different dataset baselines to evaluate how early our approach can detect packages in decline. We evaluate how early our approach can detect packages in decline based on all packages that our approach classifies as packages in decline.

1) ***npms* dataset:** This dataset was crafted from the *npms* rankings, as explained in Section IV-A. In this dataset, we start measuring the packages in decline before April 18th, 2019, where the *npms* score was still stable.
2) **Deprecated dataset:** This dataset was crafted from the deprecated `npm` packages, as explained in Section IV-C. In this evaluation, we aim to assess how far in advance we can use our approach to identify packages that have become deprecated. In this evaluation, we use the deprecation date of each package as the starting point to measure whether the package is in decline.
3) **State of JavaScript dataset:** We collect this dataset from the State of JavaScript survey of 2019 [47], as explained in Section IV-B. We label packages with a share of satisfaction less than 50% as in decline and the rest of the packages as not in decline. In this evaluation, we measure the packages in decline before November 25th, 2019, which is the date of receiving the first survey response.

Table II: Results of three datasets on how early in months our approach can detect packages in decline.

Dataset	Labeled as in decline	Classified as in decline	Time (months) Mean	Median
npms	2259	2467	18.35	12.57
Deprecated	552	446	16.15	13.29
Survey	4	3	13.13	4.80

Results. Table II presents the results of our experiment, showing how far in advance our approach can detect packages in decline. The first row in the table shows that our approach classifies 2,467 packages from the *npms* dataset as in decline with an average of 18.35 (median = 12.57) months before April 2019 (S_2). To reiterate, only after the S_2 date, these packages have shown a steep decline in the *npms* scores. Our results show that half of the packages were experiencing consistent centrality decline for more than a year before this decline was captured by the *npms* metrics.

The second row in Table II shows the results for the deprecated dataset. Our approach was able to identify the centrality decline on average more than a year (16.15 months) before the packages became deprecated. Also, the decrease in the centrality rankings captures the decline of 446 out of 552 deprecated packages. Our results indicate that the centrality trend can be used as an early indicator of deprecated packages, with a good recall, capturing 80% of the deprecated packages.

Finally, the third row in Table II shows the results of our evaluation using the State of JavaScript survey dataset. Our approach correctly classified three out of the four labeled in decline packages with an average of 13.13 (median = 4.80) months before the first survey response date without any false alarms.

Figure 8 shows the distribution of time in months for how early our approach can detect packages in decline across the three datasets. The figure shows that our approach detects 25% of packages in decline more than 31 months before the significant *npms* score decrease, and 22 months before a package got deprecated.

The results show that our approach can detect packages in decline on average **18.35 months** before the *npms* score declines. Also, it detects packages in decline on average **16.15 months** before a package gets deprecated.

RQ3: How does our approach compare to other metrics in detecting packages that are in decline?

Motivation. After determining our approach's effectiveness in detecting packages in decline, months in advance, we would like to know if other widely used metrics already capture (or complement) the information centrality indicates. There are already several metrics, e.g., as the number of GitHub stars from their repository project, that aim to provide a popularity indicator of `npm` packages and have been used by prior work, (e.g., [45, 52, 14, 53, 54, 55]). If centrality is already properly captured by other widely used metrics, there is no incentive to incorporate centrality in the current package platforms. If the centrality trends, however, provide a new perspective on the popularity and community interest of a package, there is a good motivation to make the centrality information more accessible to developers to improve their community awareness.

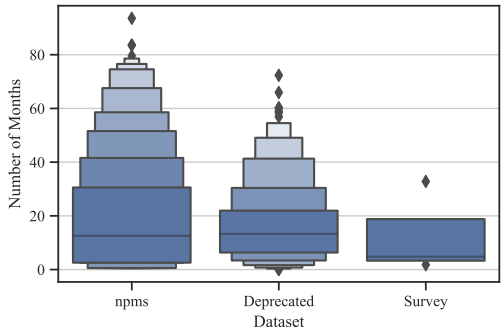


Figure 8: Letter-value plots for the distribution of how early our approach can detect packages in decline.

Approach. We are particularly interested in assessing how much of the centrality is already captured by metrics that the *npms* analyzer uses. In particular, we studied metrics that present the number of dependents, number of package downloads, Github stars, and Github forks. We evaluate if the centrality trend correlates with these metrics and whether we could use these previously used metrics to detect packages in decline, with similar or better performance than our centrality trends.

To compare our approach with the other metrics, we start by collecting the monthly number of dependents, downloads, stars, and forks of 40,619 packages in `npm`. We retrieve the number of monthly dependents using the dependency graphs we build to measure the centrality, explained in Section III-A. For the number of downloads, we use the `npm` REST API[6] to collect the daily number of downloads for the time between each package creation date (not before February 2015, which earliest data that the API keeps) until December 2020. Then we aggregate the daily downloads for every month.

The GitHub API does not provide an endpoint to retrieve the historical number of stars and forks. To overcome this challenge, we rely on the API of Porter.io,[7] a service that analyzes Github continuously and retrieves the historical number of stars and forks for a wide range of repositories. Thus, we use Porter.io's API to collect the historical number of Github stars and forks for package repositories with more than 100 stars in *npms* at December 27th, 2020. We omit packages with fewer than 100 stars, to prevent our analysis from being dominated by packages that are seldom used by the community.

After collecting the metrics for all packages with more than 100 stars, we notice that not all packages have sufficient data for our analysis. For instance, some packages lack sufficient historical data or one or more of their metrics have all the

[6]https://api.npmjs.org
[7]https://porter.io

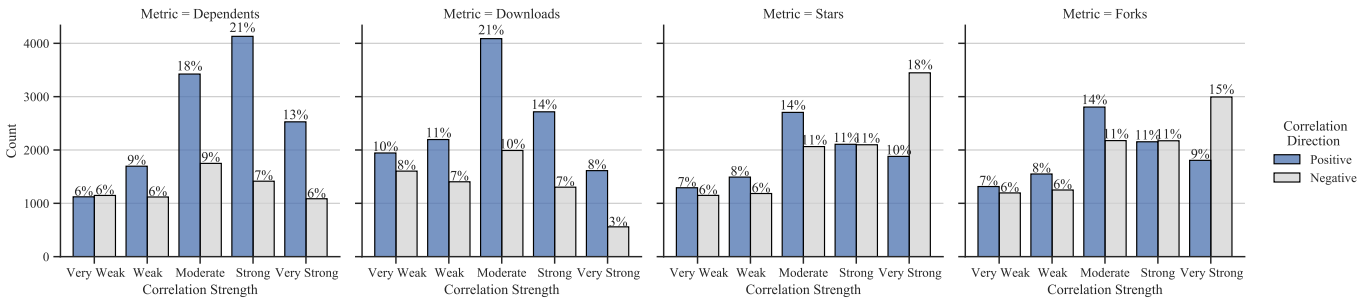


Figure 9: The distribution of the correlation between centrality and the metrics.

data points as zero, e.g., packages that have no dependents. Therefore, to simplify our analysis and report results from a uniform dataset, we exclude packages that do not have sufficient information for all metrics. This step excluded 21,201 packages from the initial set of 40,619; thus, our analysis is based on 19,418 packages.

To evaluate if the other metrics' trends indicate the same trend as centrality rankings, we test the correlation between the monthly centrality trend and each of the other metrics' monthly trends. We use Spearman's rank correlation test, and we apply the correlation test on the metrics for each package separately. We use Spearman's rank correlation coefficient since our dataset is not normally distributed [56]. Spearman's rank correlation coefficient (ρ) has a value that ranges between +1 and -1. In our context, +1 means that a metric value always increases when the centrality increases and -1 means that a metric value always decreases when the centrality increases. A Spearman (ρ) of zero indicates no correlation between the metic and the centrality [56, 57].

Results. Figure 9 shows the distribution of Spearman's rank correlation coefficient (ρ) between the centrality trend and the trend of each of the other popular metrics. Following the guidelines of Fowler et al. [57], we group the correlation distribution into five intervals: very weak correlation (0.00 to 0.19), weak correlation (0.20 to 0.39), moderate correlation (0.40 to 0.69), strong correlation (0.70 to 0.89) and very strong correlation (0.90 to 1.00). The figure plots the correlation results for 19,418 packages. We observe that centrality and the evaluated metrics have correlations that spread all the spectrum from a perfect positive correlation ($\rho = +1$) to a perfect negative correlation ($\rho = -1$). Overall, this shows that centrality is not aligned to the other metrics for most packages, indicating that centrality may provide new information that is not captured by the other metrics. Next, we discuss the comparison to each metric and its implications.

As shown in Figure 9, the dependents metric shows the strongest correlation with centrality amongst the evaluated metrics. Roughly a third of the packages (34%) have a strong or very strong correlation between its number of dependents and centrality. This is somehow expected since packages with high centrality tend to have many dependents and vice versa. Still, this strong correlation does not hold for the majority of packages that we evaluated because our approach to calculate the centrality uses an algorithm that considers not only the

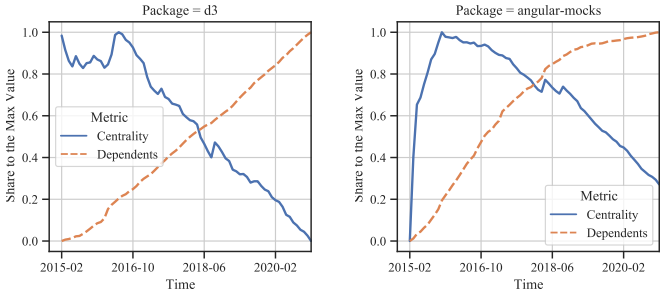


Figure 10: Examples of packages with strong negative correlation between the centrality trend and the number of dependents trend. We normalize metric values to range between 0 and 1.

number of dependents but also the importance of each of them. This explains why 13% of the packages have a strong negative correlation between the number of dependents and centrality. Such packages, such as the examples in Figure 10, have shown a steady increase in the number of dependents but an equally steady decrease in their centrality in `npm`.

The number of downloads also has a strong positive correlation with centrality in 22% of the packages. Similar to the case of the number of dependents, it is expected that packages that rise in the centrality ranking will have an increase in the number of downloads. In 36% of the cases, however, the centrality and the number of downloads are only weakly correlated (positively and negatively), and in 10% of the packages, they have shown a strong negative correlation. As shown in the Moment.js example (Section II), these are the packages that, albeit having a constant increase in their downloads, are falling in the ranking and becoming less central in the `npm` network. These are the packages in which centrality can work best as an indicator of community interest. The number of downloads depends on the number of installed systems, which may take a longer time to reflect the package's actual community interest.

The stars and forks metrics have approximately half the packages positively correlated with centrality and half the packages negatively correlated with centrality. This is a consequence of the monotonic characteristic of stars and forks. Projects tend to always increase their number of stars/forks, as contributors only rarely remove stars from a project. In fact, in our dataset only 2.39% of the packages showed a substantial

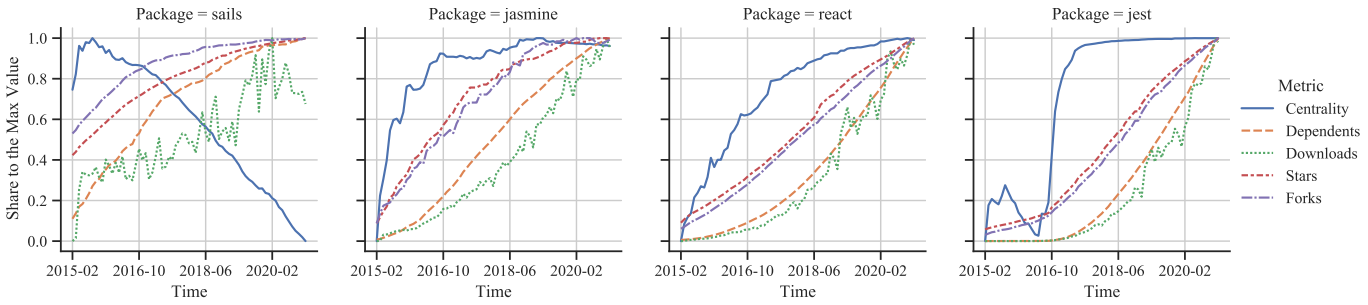


Figure 11: Line plots showing the trend of centrality alongside with the trend of other metrics. We normalize metric values using the min-max method where values range from 0 and 1 [28].

decrease in the number of stars and forks in their life cycle. Centrality, on the other hand, may increase and decrease as the community shifts its interest to the package or away from it.

To gain a better understanding of how these metrics are different, we use the following process to select four package examples: 1) We use the State of JavaScript 2019 survey that we explain in Section IV-B to select popular packages. 2) The survey includes 28 `npm` packages; we order them based on the community satisfaction score and select a package from each quartile, i.e., Sails.js, Jasmine, React, and Jest with satisfaction scores 26%, 67%, 89%, and 96%, respectively. Figure 11 plots each of the four packages with their monthly trend for all metrics.

With the decrease in maintenance activities and the increase in the number of unfixed bugs, developers start discussing the quality and health of the package Sails.js [58, 59]. We observe from Figure 11 that the package Sails.js has a decreasing centrality trend since 2015; however, all other metrics continued to increase. The centrality trend is more consistent with the survey results, where 74% of the developers (1,166 developers) that said they used the package Sails.js responded that they would not use the package again. Even though the package decreases in centrality, the package is still increasing in the number of downloads and other metrics.

Conversely, the packages Jasmine and React, which have relatively higher satisfaction scores, show a consistent increase in the centrality trend. The package Jest showed an interesting change in the centrality evolution. The package had known performance issues until mid 2016 [60], where the centrality decreased. After the maintainer of Jest performed a complete rewrite of the package to overcome its issues [61], and having these changes well-received by the community [62], Jest started showing a significant increase in centrality. By looking at Figure 11, we see that only the centrality measure captures the changes of the community's interest toward Jest.

> Centrality tends to provide trends that are different from those provided by other metrics such as dependents, downloads, stars, and forks.

VI. Tool Prototype

The main implication of our study is that reporting the centrality trends of packages as a popularity metric in `npm` can be very informative for developers. Developers should use the centrality trend, together with other popularity metrics, to have a better informed assessment on which packages to select. To enable this, we build a prototype web browser extension called `Centrality Checker` that uses our approach of detecting package in decline. Our prototype extension helps inform developers about the centrality trend when they browse a package on the official `npm` website.[8]

We build the tool as a Chrome Extension. Users can activate our extension in their Chrome browser. Once they browse a package on the *npms* website, our extension includes the package centrality trends and the result of examining if the package is in decline into the `npm` website. The initial view when a user browses a package on `npm` shows the centrality trend of the last year. Users can hover over the centrality trend chart to explore the monthly centrality ranking values from the last year. Figure 12 shows an example of an `npm` package with the proposed Chrome extension enabled. In this example, we show the package underscore ① with the centrality information embedded ②.

When a user browses a package on the `npm` website, the extension sends a request to a backend server to retrieve the needed data to render and embed the centrality ranking into the `npm` website. The backend continuously retrieves the dependency change events from the `npm` registry and calculate the centrality once every month as described in Section III-A. The backend then determines whether each package is in decline using the approach described in Section III-B. Finally, the backend caches the results to be served efficiently to our web browser extension. The tool is publicly available and can be installed through the Chrome Web Store.[9] Also, we open sourced the tool on Github.[10]

Scalability. With the exponential growth in the number of packages in the `npm` ecosystem [4], the time required to incrementally build the monthly dependency graph and calculate the centrality for all packages increases over time. In particular, as shown in Figure 13, the time required to update the dependency graph increased from 1 minute in January 2019 to 2 minutes in December 2020. The same goes for the time needed for calculating the centrality and detecting

[8]https://www.npmjs.com/

[9]https://chrome.google.com/webstore/detail/centrality-checker/bmpafkghbmojppjoeienibieljacdoaj

[10]https://github.com/centrality-checker/chrome-extension

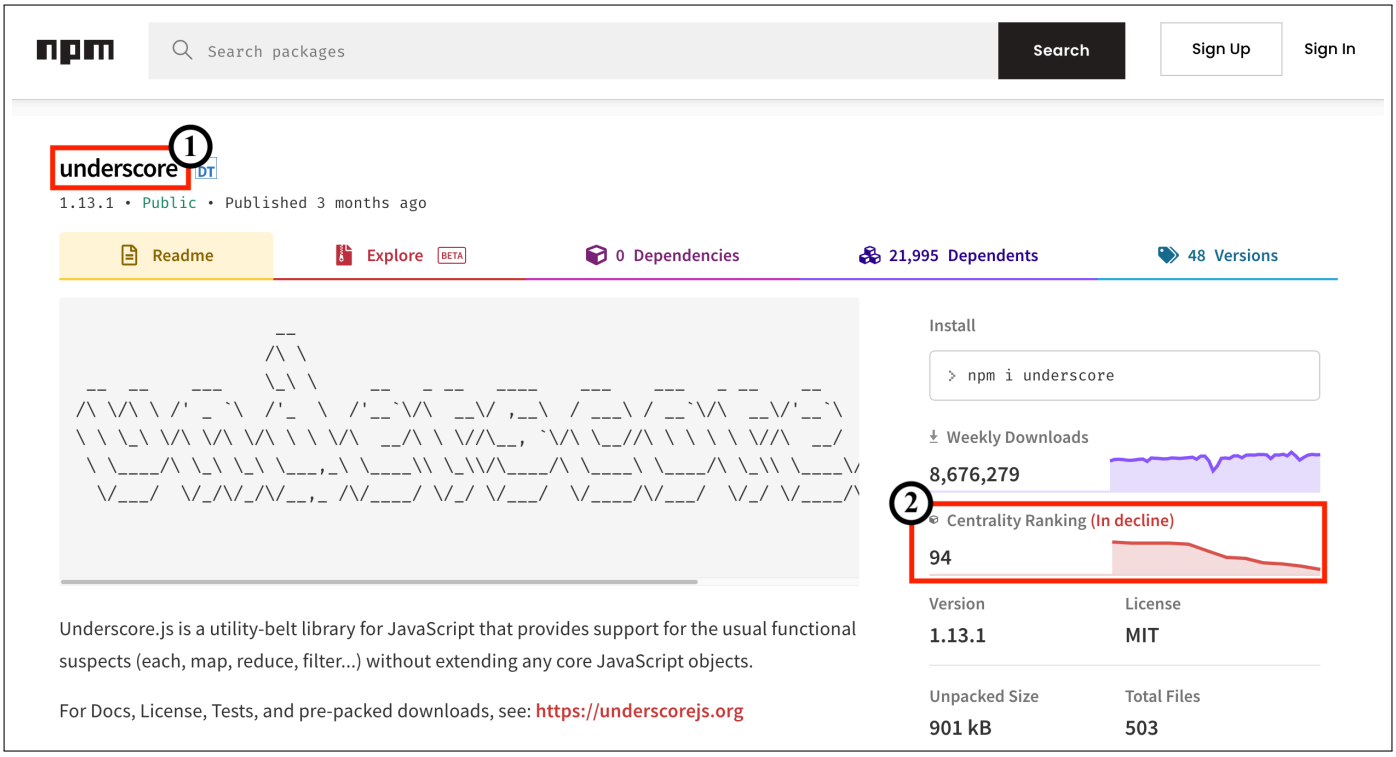


Figure 12: A screenshot of the npm website showing the package underscore with the integrated centrality information from our Chrome extension.

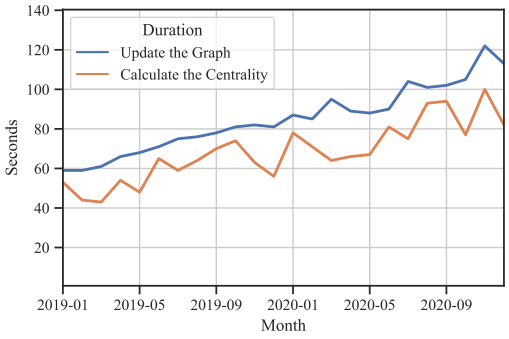


Figure 13: The time required to update the dependency graph and calculate the centrality for all packages. The experiment was performed on a conventional machine with an Intel Core i5 processor and 16GB of memory.

packages that are in decline, which increased from 50 seconds to 100 seconds. However, even with this increase, the cost of running our approach is relatively low and it can scale to handle the rapid growth of the npm ecosystem.

VII. Related Work

In this section, we discuss the work that is most related to our study.

Several studies examine the overall growth of software ecosystems. For example, Wittern et al. [6] did the first large-scale study of the npm ecosystem. They study the evolution of the npm ecosystem regarding growth and development activities. The study found that only 27.5% of packages in the npm ecosystem are depended upon, indicating that developers largely depend on a core set of packages. Decan et al. [4] also empirically compare the evolution of the dependency network in seven software packaging ecosystems. Their results show how fast each packaging ecosystem and packaging dependency network is growing over time. They observe the continuing growth of the number of packages and their dependency relationships. Some other work also studies the evolution of software ecosystems (e.g., [1, 63]). In the same line with these existing studies, our work examines the evolution of npm ecosystem in terms of its dependency graph. However, we focus on employing the npm dependency graph and calculate the centrality for each package to identify npm packages that are in decline.

Other work has been done to examine software projects that are not active anymore. For example, Coelho et al. [10] use machine learning classifiers to identify unmaintained GitHub projects. They also examine the level of maintenance activity of active GitHub projects, aiming to detect unmaintained projects. In an extension work, Coelho et al. [11] developed a metric to alert developers about the risks of depending on a given GitHub project based on the built ML classifiers. In the context of the Python ecosystem, Valiev et al. [9] studied the factors that affect the sustainability open source projects. Their results show that the centrality of a project in the ecosystem dependency network has a high impact on the project activities. Other works also investigate the overall popularity of open source projects. For example, Borges et al. [52] studied the popularity of GitHub repositories. They were able to identify four patterns of popularity growth, which

relate to factors such as stars and forks. As shown in the work mentioned above, examining the level of activity of an open source project is of critical importance, in particular, for packages in software ecosystems in order to maintain healthy dependencies. Hence, our work addresses this issue by detecting which npm packages are in decline.

There is also a body of research that investigates specific aspects of packages in a software ecosystem, including the source code size of packages [7], the impact of forks on the popularity of packages [54], conflicts between used JavaScript packages [64] or Python packages [65], identifying breaking updates in npm package [66], and studying cross-project bugs that may impact a large part of a software ecosystem [67]. Similar to these aforementioned studies, we focus on one aspect of the used packages in the npm ecosystem: the package centrality. We propose the use of package centrality to identify packages in decline and evaluate its effectiveness in the npm ecosystem.

VIII. Threats to Validity

In this subsection, we discuss threats to the validity of our study.

A. 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 npm. This limitation will impact the centrality of packages that are not meant to be reused by other packages, but other JavaScript applications. Future work should investigate how to incorporate JavaScript applications in the network and how to attribute their importance in the npm network (e.g., using the number of stars in GitHub). In our approach, the package importance is calculated by the centrality of its dependents, however, applications are not meant to be reused by other projects.

Another important threat to internal validity concerns the datasets that we used as baselines when evaluating our approach. In our baseline datasets, we used various thresholds that impact which packages to include and their labeling. Since having a gold standard for npm's community interest is very difficult, we combine evaluations made from three datasets to mitigate for the lack of a large-scale ground truth. Still, there is a need for a long term evaluation of the centrality as a complementary metric for current popularity metrics. Future work could investigate if developers find centrality a useful metric when selecting packages. 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 enhance) our approach [23].

B. Threats to External Validity

Threats to external validity are related to the generalizability of our findings. Our investigation 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, the size of packages in the npm ecosystem is relatively small compared to the size of modules and software components in other ecosystems and programming languages. The small package size in the npm ecosystem could lead to different dynamics compared to other ecosystems, which might significantly affect packages' characteristics such as the maintenance lifetime, release span, and barriers to migrate to other packages. While centrality is a commonly employed metric to evaluate the importance in highly-connected systems, such as software ecosystems, the performance of our approach might be linked to the highly dynamic characteristics of npm. Future work needs to investigate if a similar approach can also help identify packages in decline in other ecosystems such as PyPi and Maven.

IX. Conclusion

This paper presents a novel and scalable approach for using the centrality of packages to identify packages in decline. We evaluate our approach in npm, one of the largest and most popular software ecosystems. Our evaluation showed that the centrality trends were effective at identifying packages in decline (RQ1). When classifying packages as in decline and not in decline, our approach can distinguish between the two classes with an AUC of 0.9. Our approach correctly classified 87% of the packages in decline, on average 18 months before the *npms* aggregated score (RQ2). By evaluating the correlation between centrality and current popularity metrics (e.g., number of downloads), we have shown that centrality trends can provide new information, not currently captured by *npms* (RQ3). We implemented our approach in a tool that can be used by developers to complement current *npms* popularity metrics with our centrality trends. Our approach can provide a more accurate depiction of the shifts the community interest makes and help inform developers when selecting packages for their software projects.

Our paper outlines some directions for future work. First, in this paper, we use centrality as an indicator of packages in decline. We believe investigating and understanding why packages' centrality is rising or declining is critical since it helps developers make more informed decisions. Another interesting followup work is to propose an automated approach to finding future central packages so they can receive the attention needed to boost their evolution as early as possible. Finally, after identifying packages in decline, the next step should be assisting developers in replacing them. Thus, we plan to develop an approach that suggests better alternative packages for those in decline.

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] 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 Joint Meeting on the Foundations of Software Engineering, ser. ESEC/FSE '17. ACM, 2017, p. 385–395.

[3] 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.

[4] 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.

[5] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, p. 384–417, 2018.

[6] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. ACM, 2016, pp. 351–361.

[7] 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.

[8] M. den Besten, C. Amrit, A. Capiluppi, and G. Robles, "Collaboration and innovation dynamics in software ecosystems: A technology management research perspective," *IEEE Transactions on Engineering Management*, pp. 1–6, 2020.

[9] 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 Joint Meeting on the Foundations of Software Engineering*, ser. ESEC/FSE '18. ACM, 2018, p. 644–655.

[10] 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.

[11] 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.

[12] T. Dey and A. Mockus, "Are software dependency supply chain metrics useful in predicting change of popularity of npm packages?" in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE'18. ACM, 2018, p. 66–69.

[13] S. Zhou, B. Vasilescu, and C. Kästner, "What the fork: A study of inefficient and efficient forking practices in social coding," in *Proceedings of the 2019 27th Joint Meeting on the Foundations of Software Engineering*, ser. ESEC/FSE '19. ACM, 2019, p. 350–361.

[14] H. Borges and M. Tulio Valente, "What's in a GitHub star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112 – 129, 2018.

[15] 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.

[16] S. Wasserman and K. Faust, *Social network analysis: methods and applications*. Cambridge University Press, 1994.

[17] J. Wang, W. Guo, and K. Y. Szeto, "Optimization of financial network stability by genetic algorithm," in *Proceedings of the International Conference on Web Intelligence*, ser. WI '17. ACM, 2017, p. 517–524.

[18] F. Cadini, E. Zio, and C.-A. Petrescu, "Using centrality measures to rank the importance of the components of a complex network infrastructure," in *Critical Information Infrastructure Security*, R. Setola and S. Geretshuber, Eds. Springer Berlin Heidelberg, 2009, pp. 155–167.

[19] G. Stergiopoulos, M. Theocharidou, P. Kotzanikolaou, and D. Gritzalis, "Using centrality measures in dependency risk graphs for efficient risk mitigation," in *Critical Infrastructure Protection IX*, M. Rice and S. Shenoi, Eds. Springer International Publishing, 2015, pp. 299–314.

[20] W. Maharani, Adiwijaya, and A. A. Gozali, "Degree centrality and eigenvector centrality in Twitter," in *2014 8th International Conference on Telecommunication Systems Services and Applications*, ser. TSSA '14. IEEE, 2014, pp. 1–5.

[21] J. B. Hong, D. S. Kim, and A. Haqiq, "What vulnerability do we need to patch first?" in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 684–689.

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

[23] D. E. Costa, R. Abdalkareem, E. Shihab, M. A. Saied, and B. Adams, "Replication package: Towards using package centrality trend to identify packages in decline," Jan. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5003442

[24] M. Johnson-Pint, "Project status - moment.js," https://momentjs.com/docs/#/-project-status/, Sep. 2020, (Accessed on 12/30/2020).

[25] S. Nanavati, "new audit: add large-javascript-libraries audit," https://github.com/GoogleChrome/lighthouse/pull/11096, Jul. 2020, (Accessed on 01/30/2021).

[26] N. Daley, "Replace moment.js with date-fns by noelledaley," https://github.com/hashicorp/vault/pull/5789, Nov. 2018, (Accessed on 01/30/2021).

[27] P. Birchler, "Replace moment with date-fns," https://github.com/google/web-stories-wp/pull/4484, Sep. 2020, (Accessed on 01/30/2021).

[28] Codecademy, "Normalization," https://www.codecademy.com/articles/normalization, 2021, (Accessed on 09/02/2021).

[29] A. Decan, T. Mens, and E. Constantinou, "On the evo-

lution of technical lag in the npm package dependency network," in *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '18. IEEE, Sep. 2018, pp. 404–414.

[30] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *New Opportunities for Software Reuse*, ser. ICSR '18. Springer International Publishing, 2018, pp. 95–110.

[31] npm Docs, "Install packages," https://docs.npmjs.com/cli/v6/commands/npm-install, 2021, (Accessed on 01/21/2021).

[32] "registry — npm docs," https://docs.npmjs.com/cli/v6/using-npm/registry, (Accessed on 01/04/2021).

[33] 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.

[34] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, Nov. 1999, previous number = SIDL-WP-1999-0120.

[35] A. Kashcha, "npm packages sorted by PageRank," http://anvaka.github.io/npmrank/online, 2017, (Accessed on 01/15/2021).

[36] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

[37] D. F. Gleich, "PageRank beyond the web," *SIAM Review*, vol. 57, no. 3, p. 321–363, Jan. 2015.

[38] B. Manaskasemsak and A. Rungsawang, "An efficient partition-based parallel PageRank algorithm," in *11th International Conference on Parallel and Distributed Systems*, ser. ICPADS '05, vol. 1, 2005, pp. 257–263 Vol. 1.

[39] K. Berberich, S. Bedathur, G. Weikum, and M. Vazirgiannis, "Comparing apples and oranges: Normalized PageRank for evolving graphs," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. ACM, 2007, p. 1145–1146.

[40] NIST/SEMATECH, "e-handbook of statistical methods: Linear least squares regression," https://www.itl.nist.gov/div898/handbook/pmd/section1/pmd141.htm, Apr. 2012, (Accessed on 01/30/2021).

[41] G. G. Judge, W. E. Griffiths, R. C. Hill, H. Lutkepohl, and T.-C. Lee, *The Theory and Practice of Econometrics*. Wiley, Jan. 1985.

[42] C. Farrell, "Explore deprecation of istanbul-api," https://github.com/istanbuljs/istanbuljs/issues/321, Mar. 2019, (Accessed on 06/23/2021).

[43] ——, "Deprecate istanbul-api," https://github.com/istanbuljs/istanbuljs/pull/378, Apr. 2019, (Accessed on 06/23/2021).

[44] R. Benitte and S. Greif, "The state of JavaScript survey," https://stateofjs.com/, Jan. 2021, (Accessed on 12/20/2021).

[45] A. Abdellatif, Y. Zeng, M. Elshafei, E. Shihab, and W. Shang, "Simplifying the search of npm packages," *Information and Software Technology*, vol. 126, p. 106365, 2020.

[46] npms, "About npms," https://npms.io/about, (Accessed on December 26, 2018).

[47] S. Greif and R. Benitte, "The state of JavaScript 2019," https://2019.stateofjs.com/, Dec. 2019, (Accessed on 12/20/2020).

[48] F. Lindesay, "Jade node template engine - npm," https://www.npmjs.com/package/jade, Jun. 2016, (Accessed on 02/01/2021).

[49] J. Nam and S. Kim, "Clami: Defect prediction on unlabeled datasets (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE, 2015, pp. 452–463.

[50] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 485–496, 2008.

[51] F. E. James, "Monthly moving averages—an effective investment tool?" *Journal of Financial and Quantitative Analysis*, vol. 3, no. 3, p. 315–326, 1968.

[52] 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.

[53] M. Papamichail, T. Diamantopoulos, and A. Symeonidis, "User-perceived source code quality estimation based on static analysis metrics," in *2016 IEEE International Conference on Software Quality, Reliability and Security*, ser. QRS '16, 2016, pp. 100–107.

[54] J. Zhu, M. Zhou, and A. Mockus, "Patterns of folder use and project popularity: A case study of GitHub repositories," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. ACM, 2014.

[55] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the diversity of software package popularity metrics: An empirical study of npm," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER '19, 2019, pp. 589–593.

[56] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938.

[57] J. Fowler, L. Cohen, and P. Jarvis, *Practical statistics for field biology*. John Wiley & Sons, 2009.

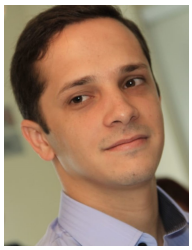
[58] Hacker News, "Status of Sails.js," https://news.ycombinator.com/item?id=10819583, Dec. 2015, (Accessed on 06/25/2021).

[59] ——, "Is Sails.js dying?" https://news.ycombinator.com/item?id=10755557, Dec. 2015, (Accessed on 06/25/2021).

[60] Jest Blog, "JavaScript unit testing performance," https://jestjs.io/blog/2016/03/11/javascript-unit-testing-

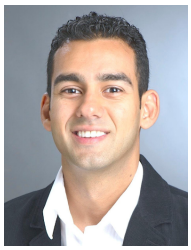
performance, Mar. 2016, (Accessed on 06/24/2021).
[61] ——, "2016 in Jest," https://jestjs.io/blog/2016/12/15/2016-in-jest, Dec. 2016, (Accessed on 06/24/2021).
[62] Hacker News, "Jest: Painless JavaScript testing," https://news.ycombinator.com/item?id=13128146, Dec. 2016, (Accessed on 06/24/2021).
[63] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *2013 17th European Conference on Software Maintenance and Reengineering*, Mar. 2013, pp. 243–252.
[64] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: Finding and understanding conflicts between JavaScript libraries," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, 2018, p. 741–751.
[65] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: monitoring dependency conflicts for Python library ecosystem," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 125–135.
[66] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. ACM, 2020, p. 466–476.
[67] W. Ma, L. Chen, X. Zhang, Y. Feng, Z. Xu, Z. Chen, Y. Zhou, and B. Xu, "Impact analysis of cross-project bugs on software ecosystems," in *2020 IEEE/ACM 42nd International Conference on Software Engineering*, ser. ICSE '20. IEEE, 2020, pp. 100–111.

Suhaib Mujahid is a Ph.D. candidate in the Department of Computer Science and Software Engineering at Concordia University. He received his master's in Software Engineering from Concordia University (Canada) in 2017, where his work focused on detection and mitigation of permission-related issues facing wearable app developers. He did his Bachelors in Information Systems at Palestine Polytechnic University. His research interests include software ecosystems, machine learning on code, software quality assurance, mining software repositories and empirical software engineering. You can find more about him at https://suhaib.ca.



Diego Elias Costa is a postdoctoral researcher in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD in Computer Science from Heidelberg University, Germany. His research interests cover a wide range of software engineering and performance engineering related topics, including mining software repositories, software ecosystems, and performance testing. You can find more about him at http://das.encs.concordia.ca/members/diego-costa.

Rabe Abdalkareem is an assistant professor in the School of Computer Science at Carleton University. He received his PhD in Computer Science and Software Engineering from Concordia University. His research investigates how the adoption of crowd-sourced knowledge affects software development and maintenance. Abdalkareem received his master's in applied computer science from Concordia University. His work has been published at premier venues such as FSE, ICSME, and MobileSoft, as well as in major journals such as TSE, IEEE Software, EMSE and IST. You can find more about him at https://rabeabdalkareem.github.io.



Emad Shihab is an associate professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queens University. Dr. Shihab's research interests are in Software Quality Assurance, Mining Software Repositories, Technical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and ACM. More information can be found at http://das.encs.concordia.ca.

Mohamed Aymen Saied is an assistant professor in the department of Computer Science and Software Engineering at University of Laval, Quebec, Canada. Prior to that, he was a Post-Doctoral Fellow in the Electrical and Computer Engineering Department of Concordia University. He received his PhD from University of Montreal. His research interests are at the intersection of software engineering, software data analytics and cloud-native systems. You can find more about him at https://saiedmoh.github.io.

Bram Adams is an associate professor at Queen's University. He obtained his PhD in 2008 at Ghent University's GH-SEL lab (Belgium). His research interests include software release engineering, mining software repositories, and the role of human affect in software engineering. His work has been published at premier software engineering venues such as EMSE, TSE, ICSE, FSE, MSR and ICSME, and received the 2021 MSR Foundational Contribution Award. In addition to co-organizing the RELENG International Workshop on Release Engineering from 2013 to 2015 (and the 1st/2nd IEEE Software Special Issue on Release Engineering), he co-organized the SEMLA, SoHEAL, PLATE, ACP4IS, MUD and MISS workshops, and the MSR Vision 2020 Summer School. He has been PC co-chair of SCAM 2013, SANER 2015, ICSME 2016 and MSR 2019, and will be ICSE 2023 software analytics area co-chair. More information can be found at https://mcis.cs.queensu.ca/bram.html.