Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies?

Samim Mirhosseini
North Carolina State University
Raleigh, NC, USA
smirhos@ncsu.edu

Chris Parnin
North Carolina State University
Raleigh, NC, USA
cjparnin@ncsu.edu

***Abstract*—Developers neglect to update legacy software dependencies, resulting in buggy and insecure software. One explanation for this neglect is the difficulty of constantly checking for the availability of new software updates, verifying their safety, and addressing any migration efforts needed when upgrading a dependency. Emerging tools attempt to address this problem by introducing automated pull requests and project badges to inform the developer of stale dependencies. To understand whether these tools actually help developers, we analyzed 7,470 GitHub projects that used these notification mechanisms to identify any change in upgrade behavior. Our results find that, on average, projects that use pull request notifications upgraded 1.6x as often as projects that did not use any tools. Badge notifications were slightly less effective: users upgraded 1.4x more frequently. Unfortunately, although pull request notifications are useful, developers are often overwhelmed by notifications: only a third of pull requests were actually merged. Through a survey, 62 developers indicated that their most significant concerns are breaking changes, understanding the implications of changes, and migration effort. The implications of our work suggests ways in which notifications can be improved to better align with developers' expectations and the need for new mechanisms to reduce notification fatigue and improve confidence in automated pull requests.**

I. Introduction

Developers are often occupied by the primary task of writing new features or fixing known defects, and as a result, often ignore or delay important but secondary preventive maintenance tasks [1]. One such task, updating software dependencies, includes upgrading the versions of the many libraries, tools, and packages that a program depends on [2]. The consequences of neglecting to update out-of-date dependencies can be severe, as dependencies can suffer from numerous security issues [3] and buggy features [4]. For instance, *Using Components with Known Vulnerabilities* [5] is listed as a top 10 application security risk in 2017 by OWASP.

Unfortunately, updating dependencies can be a time consuming task [6]—large parts of code may need to be verified and migrated to make use of new interfaces and changed functionality before using an updated library. For example, updating the mongodb driver for node.js from `2.0.x` to `2.1.x` affected the way order by parameters are used in the *sort* function. Previously, parameters were allowed to be specified as list of objects: `[{publish_date: -1}]` but in the newer version, list of lists must be provided: `[["publish_date", -1]]`. Upgrading a version of a library that contains incompatibilities in data structures, signature changes to API calls, or behavior breaking changes [7] can result in additional effort to address these changes. If a developer fails to understand the nature of a dependency change or perform insufficient testing, they can introduce undetected faults in code. Such factors may cause a developer to become reluctant to update dependencies. But, if they delay updating code too long, they may be locked out of being able to use important new features only available in new versions, as it becomes more and more difficult to adapt their code.

An important challenge is to convince developers that they should upgrade despite all the associated difficulties and risks. One way to convince developers they should update is to provide incentives that encourage them to update. For example, *badges* are images that can be displayed on a Github project's profile page. External tools can update the images to reflect the status of the project, such as build status or code coverage. A popular tool, *David-DM*[1], will check if a project's dependency is out-of-date and display a red version of the badge if it is: dependencies out of date. In the context of open source systems, these badges can signal [8] to potential contributors or users that the project uses practices such as continuous integration (CI) and keeping its dependencies updated. Given the importance of these signals in perceived quality [9], project owners have a high incentive to maintain "green" badges.

Another way to encourage developers to upgrade is to reduce the risk and effort involved through *automated pull requests*. For example, several types of software engineering bots [10] have emerged as automated mechanisms that can help developers automate routine tasks, such as report code coverage. A popular tool, *greenkeeper*[2], is a bot that can help keep a project's dependencies up-to-date by automatically updating versions of a dependency in a package management file, such as maven (pom.xml) or npm (package.json) and creating a pull request with the change. Using another bot, such as Travisbot[3], the automated pull request can further trigger a continuous integration build and verify that no breaking changes are identified by the passing test suite. With the combined contributions of these bots, a developer may be

[1]https://david-dm.org/
[2]https://greenkeeper.io/
[3]https://travis-ci.org/

able to gain confidence in performing an upgrade and more quickly integrate a suggested upgrade.

Despite the importance of dependency management, there is little empirical evidence supporting the effectiveness of using mechanisms such as badges and automated pull requests. Badges support awareness and create pressure but do not provide actionable results. Automated pull requests support actionable results, but factors such as notification fatigue [10] and insufficient test coverage may reduce the effectiveness of these requests. In this paper, we analysis 7,470 Github projects, split into projects that exclusively use David-DM (badges), greenkeeper (automated pull requests), or a baseline of neither. We then survey developers on issues related to dependency management and their experience using dependency management tools and greenkeeper. We find that projects that use automated pull requests upgrade dependencies 1.6x over baseline projects, and projects with badges upgrade 1.4x over baseline projects. The speed of updates is also faster for automated pull requests (1.31x over badge, 1.33x over baseline). Further, specific dependencies and their versions, such as mongodb (version `0.7.5`–`2.2.26`), are more likely to be at their highest version (`2.2.26`) in projects using automated pull requests. These finding are interesting in light of results that find social factors tend to outperform technical factors in predicting time to evaluate a pull request [11].

There were also several surprising results. For automated pull requests, only 32% were merged (compared with 80% of typical pull requests merged within 3 days [12]). Further, several merged automated pull requests were often immediately downgraded back after 2–3 days. For projects using continuous integration to automatically build code in a pull request, the merge rate was improved, but only marginally so (26% merged without CI and 32% merged with CI). Finally, there is evidence that developers' suspicion of breaking changes is not unfounded: 24% of builds fail when changing a dependency version. This evidence provides guidance both for developers establishing risk for performing an upgrade and for researchers working on automated API migration [13].

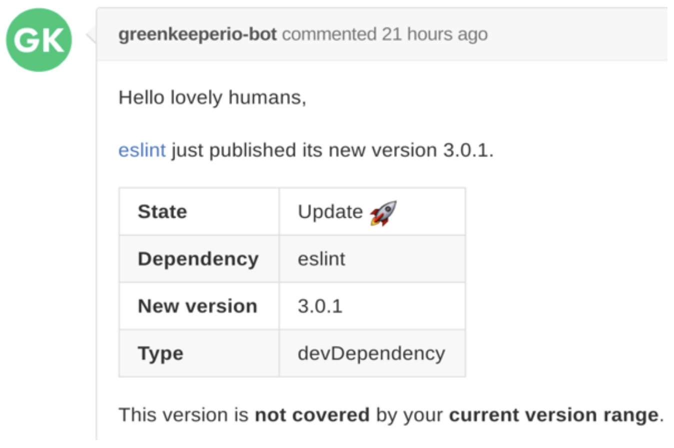
Finally, developers provided several insights into their practices and behaviors associated with dependency management. Developers were mixed in their preference for passive mechanisms such as dashboards and badges versus automated mechanisms. Developers also reported several concerns related to breaking changes and understanding the implications of changes before merging. This suggests that automated pull requests need mechanisms for improving arguments used to convince developers about the benefits offered by a particular upgrade and improve confidence in the safety of the upgrade.

This paper makes the following contributions:

- The first empirical study that compares the stimulating effects of automated pull requests and social pressure created by badges on updating dependencies.
- A qualitative analysis of developers' experiences with using automated pull requests, badges, and dependency management strategies.
- Implications for improving the design of automated pull requests for software engineering tasks.

II. Motivating Example

Vera maintains a popular open source library that uses over 50 software dependencies, but because of a full-time job and other commitments, she only has time once a month to make improvements to the software. Recently, a developer created a new issue in which he complained that Vera's project used many old dependencies that was causing him problems in upgrading to a new version of NodeJS. Soon after, many other developers piled on, complaining about several security vulnerabilities that were present in these dependencies as well. Distraught about all the negative feedback, Vera wants to be more proactive in managing her software dependencies, but she does not know where to start.



greenkeeperio-bot commented 21 hours ago

Hello lovely humans,

eslint just published its new version 3.0.1.

State	Update 🚀
Dependency	eslint
New version	3.0.1
Type	devDependency

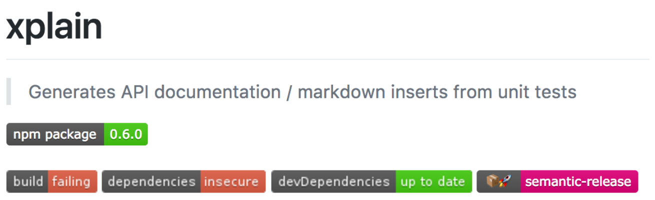
This version is **not covered** by your **current version range**.

Fig. 1. Automated pull request created by greenkeeper.io

One developer recommends she try out greenkeeper.io. He explains how it works:

- Whenever a new version is available for a package, a bot makes a pull request to a project's repo.
- In the pull request, the package management file, package.json is changed to have the new version, e.g., `2.3.4 => 2.3.5`
- An additional information is included in the comment of the pull request.
- Since, Vera is using a continuous integration server (Travis CI) to run on pull requests, then the pull request will trigger a build and testing step in order to verify if the version change will break her code.
- If Vera wants to upgrade, she simply merges the pull request, otherwise she closes the pull request to ignore the recommendation.

Vera is interested in adopting greenkeeper.io, but worries that it might cause too many distractions and that she would eventually just ignore the notifications. Another developer suggested that she simply use DavidDM. The developer argues that because it is just a simple badge displayed in her project's README.md, she can check the site occasionally to see if any dependencies should be upgraded. Further, anyone visiting her Github project would see that her project is using out-of-date dependencies, which creates additional social pressure. If more information is needed, she can visit the site and see a dashboard describing out-of-date and insecure dependencies.


xplain

Generates API documentation / markdown inserts from unit tests

npm package 0.6.0

build failing | dependencies insecure | devDependencies up to date | semantic-release

Fig. 2. Badges displayed on a Github project's profile page

Vera likes the simplicity of DavidDM, but she worries that without any proactive reminder, she might forget to check the site and worse, she still needs to manually verify if a new version will work with her library. Vera wishes there was some evidence that would demonstrate the effectiveness of the different approaches. Further, she would like to catalogue the strengths and weaknesses of each tool in order to design a better dependency management tool, as another side project.

III. Methodology

To learn whether dependency management tools help developer—and if they do, to what extent—we extracted data from five sources: (i) GitHub's public dataset on Google BigQuery, which is the complete content of public repositories on GitHub; (ii) GitHub Archive dataset on Google BigQuery, which is a project that collects all the public events on GitHub; (iii) Travis-CI API; (iv) GitHub API; and (v) Git history of public projects on GitHub. We will use this collected data to answer the following research questions:

A. Research Questions

- RQ1. How does the availability of a dependency management tool influence update behavior?
- RQ2. How often are projects using the latest versions of dependencies?
- RQ3. How effective are automated pull requests?
- RQ4. How do developers perceive out-of-date dependencies and make dependency management decisions?

B. Project Selection

We select projects from the Node.js/npm ecosystem, which has been previously studied in other dependency management research [14], [15], [16]. Npm contains over 350,000 packages (currently the largest package registry in the world[4]). Npm packages have far more transitive dependencies than CRAN or RubyGems packages [15], meaning there is more risk in breaking changes. Given the dynamic nature of Javascript, and fast growing ecosystem, the problem of maintaining and updating npm packages is particularly difficult—as confirmed the top concern by a recent survey of Node.js developers [16]. Collectively, these factors provide a strong motivation for understanding how to improve this problem.

To be able to compare the effect of different dependency management tools, we defined three groups of projects: (1) control group, which does not use any dependency management tool; (2) pull request notification group, which contains projects that used greenkeeper.io; and (3) badge notification group, which contains projects that used DavidDM.

1) Obtaining Pull Request Group: We used GitHub's public dataset to select public JavaScript repositories on GitHub. GitHub Archive dataset contains all of events on public repositories on GitHub. We used a query to search for PullRequestEvents in a repository that were created by greenkeeper.io. We then selected projects that have received at least one pull request from greenkeeper.io to be included in the pull request group.

2) Obtaining Badge Group: To display a badge for a GitHub project, an image element can be added to a project file, such as in the README.md, embedding an url image obtained from DavidDM[5]. A project can get a url from badge by registering their project. Using Google BigQuery, we could search the GitHub Archive dataset for Javascript projects that contained a badge in the README.md or other projects files. We matched urls that contained references to david-dm.org, and extracted the organization (or user) and project name. We then verified that the badge actually belonged to the GitHub project. For instance, sometimes manually cloned projects contained badge references that did not belong to the project. We excluded these urls from our analysis. We then used this data to select projects for our badge group if they contained a validated badge url.

3) Obtaining Control Group: To determine control projects, we selected projects from the GitHub Archive dataset, but did not belong in the pull request group. Similarly, we used another query to verify that our control group projects did not contain badge urls.

Finally, for all groups, we used the GitHub API to collect number of stars, watchers, open issues and whether or not a project is a fork for another project. As a result, we collected 26,324 Javascript projects, which is shown in Table I.

TABLE I
Initial Projects: GK (Pull Requests), DM (Badges), CL (Control)

Groups	Repos	stars	watchers	forks	open issues	commits
GK	5488	170.3	10.4	22.1	18.9	145.9
DM	11059	111.6	7.9	27.3	8.6	98.9
CL	9777	25.4	3.3	6.0	1.5	280.3

To further refine the projects in our dataset, we used the criteria of excluding forked, non-starred, and deleted projects, and selected projects with at least 20 commits. We purposely did not want to restrict our projects to the most popular and

[4]https://www.linux.com/news/event/Nodejs/2016/state-union-npm

[5]Example badge url: https://david-dm.org/bower/bower/status.svg

very active projects. One simple reason is that many projects can be like Vera's, which can be enhanced infrequently but actively used, while still needing security and dependency updates.

As a result, we obtained 7,470 projects (see Table II). From these projects, 3619 were using badge notification, 2,578 were using pull request notification, and 1,273 did not use any tool and belong to the control group. Interesting, badge projects remained the most popular strategy, which can partly be explained by ease of setup.

TABLE II
FILTERED PROJECTS: GK (PULL REQUESTS), DM (BADGES), CL (CONTROL)

Groups	Repos	stars	watchers	forks	open issues	commits
GK	2578	360.4	19.4	43.8	20.8	238.1
DM	3619	337.2	20.5	60.0	17.0	250.7
CL	1273	185.6	13.6	37.9	8.0	205.0

C. API Selection

Because some projects can contain very different types of dependencies, we want to select a subset of dependencies that we could analyze across all projects groups.

To better isolate the effects of updating, we also looked at a set of specific APIs used commonly across many of the projects, in order to observe how updates varied across the conditions. We performed *purposive sampling*, or non-probabilistic sampling, on npm packages. We stratified selection across different levels of popularity, change deltas, and purpose. For change deltas, we made a list of APIs that usually have the most changes in each release. Major releases in some of these packages can be harder update for developers to update as they may need to rewrite large parts of their code in order to be able to update to new version. For purpose, we wanted to select several packages used in different situations, such as utility packages like `lodash`, and compare them with other packages, such as UI frameworks like `express`. The final list of packages that we selected includes `lodash`, `express`, `mongodb`, `react`, `request`, `mocha`, and `redis`.

D. Data Collection: Version Changes, Pull Requests, Builds

In order to record all dependency changes of all projects, we performed the following steps. First we clone each project from GitHub. Then we searched for the changes to 'package.json' in the source control history. Then we reviewed those changes to determine if it resulted in upgrade, downgrade, or no version change (e.g., changes to the meta-data) of dependencies. Finally, for each dependency we recorded timestamp of the version change event along with the determined type of change (i.e., upgrade, downgrade). To make this workflow scalable we automated these steps using Ansible scripts. [6]

To obtain pull requests, we examined the pull request events in Github Archive made by greenkeeper.io, and then

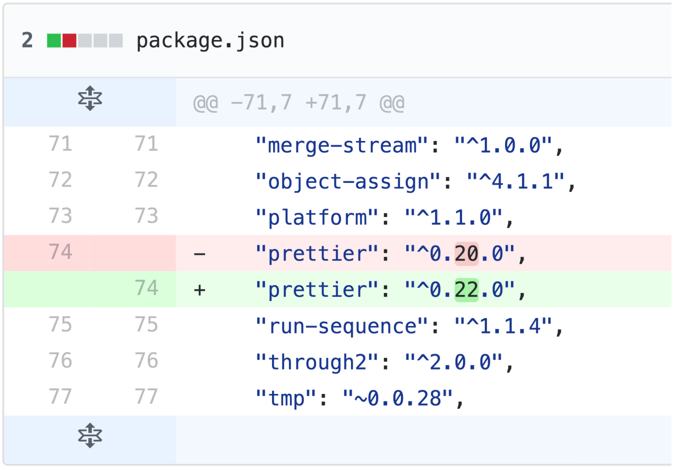
[6]https://ansible.com/

we used the GitHub API to collect additional data about the pull requests, including comments, and the resolution status (merged, open, closed).

To obtain build status, we first randomly inspected the comments of those greenkeeper.io created pull requests and we found several projects were using Travis CI, a continuous integration service which can integrate with GitHub. These projects had configured Travis CI to automatically build the project upon receiving a new pull requests. Further, a recent study of CI on Github [17] found that Travis CI was by far the most popular CI tool in use. Therefore, we focused our collection on getting the status of Travis CI builds triggered by greenkeeper.io pull requests. We used the Travis CI API to find the build status of each greenkeeper.io pull request. We could then determine whether a build failed or succeeded for each pull request.

E. Version Analysis

After collecting the data, the primary analysis involved determining whether a package was updated or not. We obtained every commit to a project's package.json and automatically analyzed differences (see Figure 3). We extracted the change packages and version pairs, in addition to other information, including the commit sha and timestamp.



```
2 ■■□□□ package.json
@@ -71,7 +71,7 @@
71  71      "merge-stream": "^1.0.0",
72  72      "object-assign": "^4.1.1",
73  73      "platform": "^1.1.0",
74      -   "prettier": "^0.20.0",
    74  +   "prettier": "^0.22.0",
75  75      "run-sequence": "^1.1.4",
76  76      "through2": "^2.0.0",
77  77      "tmp": "~0.0.28",
```

Fig. 3. Updating a dependency in `package.json`

1) Calculating Version Range Changes: From the version pairs, we determined whether there was a upgrade, downgrade, or no change. Calculating downgrades and upgrades, such as `2.3.1 => 3.0.0` can be straightforward. However, in some cases, the semantic version specification also allows a ranges of versions to be specified. For example, specifying `1.0.x`, means any patch changes are allowed. So if a package version `1.0.9` is pushed to the npm repository, then any clients that run `npm update` and have a lower version, will get the new version.

If the version range is changed, it can be changed to allow for more version ranges. For example, if there is a change from `1.0.x => 1.x`, then we say there is an increase in the range of versions allowed and we consider this an *upgrade*.

Conversely, if the change was from `1.x => 1.0.x`, then we consider this a *downgrade*. We have created a version comparison tool to account for all the ranges allowed in the semver specifications.

In total, we obtained 344,918 upgrades and downgrade events.

F. Survey

We constructed a survey with questions related to developer's experiences with dependency management and their perspectives on tool design.

1) Design: We asked several background questions related to participants' programming language use, project type, and use of package managers. Participants were also asked to describe an experience related to using an out-of-date dependency, and an experience related to a breaking change after performing an upgrade.

Participants then selected their primary concerns related to managing dependency and their strategy for managing dependencies. They were asked about their preferences related to several possible tools and design choices such as notification style (including passive displays such as badges or active devices such as automate pull requests). We also collected feedback about advanced features, such as auto-migration. Finally, participants had the option of providing general feedback in an open-response question.

A second targeted survey was designed to gather feedback from greenkeeper.io users. Our goal was to gain additional insight into some of the observations we have made in our initial analysis of our research questions. The questions are similar to the general survey, except they are tailored to experiences and design choices related to using greenkeeper.io.

2) Distribution: We distributed the survey to the general developer population. We posted to programming forums and sent to contacts at several large corporations. For our targeted survey, we sampled random projects from our pull request dataset, located pull requests created by greenkeeper.io that were closed by a developer, and then sent them tailored emails to invite them to give feedback about their experience using greenkeeper.io.

3) Analysis: We received data from 55 developers, and 7 greenkeeper.io users. To analyze the survey data, we qualitatively coded the answers pairs over multiple interactions, first obtaining codes in open coding, and then using axial coding to identity themes associated with experiences, practices, and barriers related to dependencies. During the qualitative coding process, we employed a technique of *negotiated agreement* [18] to address issues of reliability. Using this technique, the first and second authors collaboratively code to achieve agreement and to clarify the definitions of the codes. We coded the first 20% of surveys using the negotiated agreement technique, and then independently coded the remaining messages.

G. Replication

We have created a full replication package that can automatically configure a new virtual machine to install all necessary analysis tools. We have also released a standalone npm package for our semantic version range comparison tool. To support verifiability, we provide supporting online materials containing dataset, as well as the intermediate analysis which led to our findings, which other researchers may verify.[7]

IV. Results

In this section, we present result of our analysis on the data that we collected, to answer our research questions.

A. RQ1: How Does The Availability of a Dependency Management Tool Influence Update Behavior?

To understand how automated pull requests and badges may influence upgrade behavior, we examine the frequency of dependency changes. Specifically, we can examine the number of version updates, downgrades, and the time between updates. Because it may be possible for a large portion of upgrades to later be reverted to the original version, we also measure the *effective upgrade rate*. That is, we look for downgrades events for a package, and cancel the last upgrade event for the package (effectively subtracting downgrades from total upgrades). We measure these values for the groups overall, as well as for individual packages. Finally, we performed a two-tailed Wilcoxon rank-sum test on every pair in order to determine if any differences are significantly significant.

Table III displays our results.

1) Updates, Downgrades: We find that on average, automated pull requests had statistically significant higher number of updates comparing to Badges and our baseline. For example, `lodash` had 55% more update events in automated pull requests compared to Badges. Similarly, `lodash` was updated 168% more times compared to our baseline.

At the same time, automated pull requests had statistically significant more downgrades comparing to the other two groups. Overall, automated pull requests performed 10% more downgrades than Badges and 35% more downgrades than our baseline. When we inspected the time between upgrades and downgrades for automated pull requests, the average lag was a 2–3 days. This suggests that developers using automated pull requests updated faster, but they often had to downgrade quickly, because their code failed to function correctly. We discuss the consequences of these high downgrade rates later in the paper.

Finally, we compared the effective upgrade rates with the simple update measure. In general, there was a strong agreement between effective upgrade rates and the simple update measure. For example, the difference between the GK vs. control average upgrade rate (1.58x) and average effective rate (1.5x) is 0.08x. The reason why is that compared to the frequency of upgrade events, downgrades occur at a rate that is at least a order of magnitude smaller. For example, for the package `lodash`, there were 5,436 upgrade events but only 636 downgrades. For automated pull requests, there were 136,459 upgrade events, but only 22,063 downgrade events.

[7]https://github.com/alt-code/Research/tree/master/VersionBot.

TABLE III
STATISTICAL SIGNIFICANT DIFFERENCES (P <0.01) OF EFFECTIVE UPGRADES (E-UP), UPDATES, DOWNGRADES, AND UPDATE LAG (IN DAYS) BETWEEN DIFFERENT PROJECT GROUPS.

	PULL REQUESTS[1]				BADGES				CONTROL			
	e-Up	Updates	Downgrades	Lag	e-Up	Updates	Downgrades	Lag	e-Up	Upgrades	Downgrades	Lag
`lodash`	🟩 5.61	🟩 6.18	2.39	🟩 67	3.78	3.99	1.41	108	2.24	2.31	1.24	142
`mongodb`	5.35	5.62	1.71	145	3.73	4.02	1.38	123	2.22	2.89	2.33	131
`react`	🟩 4.75	🟩 5.40	1.94	62	3.79	4.14	1.65	78	2.31	2.48	1.12	84
`request`	3.91	🟨 4.51	1.70	🟨 113	🟨 4.13	4.49	1.70	119	2.67	2.68	1.10	98
`mocha`	🟨 3.29	🟩 3.73	1.67	🟩 123	3.15	3.34	1.40	151	2.00	2.16	🟨 1.19	173
`redis`	3.27	3.43	1.13	155	2.55	3.17	1.75	148	1.67	1.86	1.33	270
`express`	2.59	3.05	1.87	127	🟨 3.13	🟨 3.31	1.39	165	1.85	2.02	1.42	217
OVERALL	🟩 3.24	🟩 3.56	1.61	🟩 85	2.91	3.11	1.47	113	2.16	2.26	🟩 1.19[2]	114

🟩 GREEN indicates statistical difference between all conditions, placed at the best performing group. 🟨 YELLOW indicates statistical difference with CONTROL.
[1] PULL REQUESTS predominately have statistically higher update rates and lower update lag times compared to BADGES and CONTROL.
[2] CONTROL has a much lower downgrade rate than other groups. Pull requests can increase the need to rollback broken updates.

While we observe a difference in downgrade rates across groups, the size is still too small to impact the overall upgrade rate.

2) Update Lag: We found that on average, automated pull requests had statistically significant shorter update lag comparing to the other two groups. They had about 24% shorter lag between dependency updates comparing to the badge group, and badge group's projects had only 1% shorter lag between dependency updates in contrast to control group. For example, update lag for `lodash` was 66 days, 108 days, 142 days in pull request, badge notification and control group respectively.

The results find that both tools had positive effect on update frequency and developer behavior. pull request group had higher number of dependency updates comparing to badge, and this is because pull request will ask developer to update for all available new dependency versions. But comes at a cost: higher rollbacks.

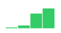
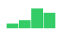
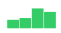
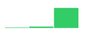
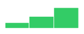
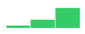
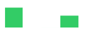
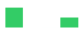
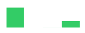
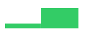
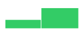
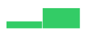
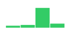
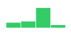
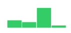
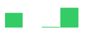
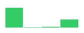
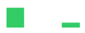
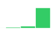
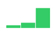
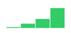
B. RQ2: How Often Projects Are Using the Latest Versions of Dependencies?

Distribution of dependency versions in each group is one of the metrics that can help us measure effectiveness of dependency management tools. For example, if 90% of projects in tool "A" use the latest version of `lodash` vs. only 50% of projects in tool "B", then it provides evidence that tool "A" was more effective. To find the distribution of dependency versions we created a bucket for each major version and incremented count of that bucket when a projects used that major version. For example if projects used versions between 1.1.1 and 3.4.0, we would create three buckets 1,2 and 3 and add projects to their corresponding bucket and then count number of projects in each bucket. Then, we compared the number of projects that used that major version in each group using a χ^2 with Yates correction.

We display our results in Table IV. For the packages we measured, automated pull request projects had the highest percentage of latest versions. For example, 50.4% of the automated pull request projects used `lodash` version 4, vs. 31.3% for badges, and 29.5% for our baseline. For `mongodb` version 2, was used by 87.5% of automated pull request projects, vs. 54.1% for badges, and 63.64% for our baseline.

In many cases, automated pull requests had between 20–50% higher occurrences of the latest version of a package over our baseline. Badges often had smaller to no difference in latest versions over our baseline.

TABLE IV
STATISTICAL SIGNIFICANT DIFFERENCES (P <0.01) IN DISTRIBUTIONS OF MAJOR VERSIONS ACROSS PROJECT GROUPS.

Package	p-value	PULL REQUESTS	BADGES	CONTROL
lodash	<.01			
mongodb	.18			
react	.71			
request	.04			
mocha	<.01			
redis	.13			
express	<.01			

C. RQ3: How Effective Are Automated Pull Requests?

To understand the effectiveness of automated pull requests, we want to find how many of pull requests were accepted. We also want to evaluate whether other factors such as availability of automated builds influenced acceptance rates. Finally, we examined the targeted developer survey to understand how developers perceived the effectiveness of greenkeeper.io.

Only 1/3 of pull requests were merged. To find how many of pull requests were merged, we ran a SQL query to find ratio of total number of pull requests by greenkeeper, to the pull requests that were merged. We also ran SQL queries on the build history of pull request group's projects which we collected from Travis CI before. 72% of projects in our initial pull request group's sample used Travis CI to automatically build all new pull request. Results show that only 24% of pull requests were failing. Pull request that had a Travis CI build had slightly higher merge rate. Specifically, 32% of pull requests that had a Travis CI build were merged, while only 27% of pull requests that didn't have a build were merged. This indicates using CI can reduce the change of encountering a broken build, but not substantially.

Surprisingly, 2% of failing builds were still merged. In the surveys with greenkeeper.io users, developers explained that they sometimes merged broken pull requests in order to manually perform more extensive testing.

1) Developer Feedback: Based on our targeted survey, we were able to better understand how developers used Greenkeeper.io.

- **Batching.** Greenkeeper.io sends a pull request per dependency update. However, several participants rather have these updates batched into a single pull request, which they could manually review in a more through manner. Several participants mentioned that npm package authors can be "trigger happy". They publish several releases in a row as they are refining a new feature or release, which results in *microbursts*, meaning that the whole community can receive 3–5 pull requests in a row, instead of just one. For example, recently a new version of `request 2.73.1` was updated several times in a few minutes, with commit messages like, "oops. forgot to remove debug", quickly followed by another version change to "remove trailing comma".
- **Content.** Participants had several suggestions on how to improve the content of the notification in the pull request body. Some developers wanted more information, such as changelog entries instead of just git commit messages. This way they could better judge the scope of the version change. However, one participant felt the content was "too busy", and should be limited to dependency, version, and whether it passes or fails CI. However, the participant did also suggest the possibility of including a *confidence score* that helped estimate the risk of merging a pull request. Future work could identify possible factors, such as rollbacks or number of signature changes, in support of a recommender system for merging pull requests. Finally, one developer wanted more control over how to customize the style of the commit messages, which did not fully follow the project's conventions.
- **Cleanup.** There were several concerns related to improving the workflow and aspects not fully automated. For example, many branches can be created from the pull requests, but they are not automatically removed after being merged. This can result in developers spending significant time manually cleaning up and verifying which branches can be removed and which ones may correspond to open requests. Another task was related to publishing a new release. Currently, if a developer merges a pull request, they still have to manually create a new release. Even if there is simple change, extra work is involved in publishing a new release for users.

Based on our results, we find that automated pull requests can be effective but noisy. Previously, we found that they can result in higher and faster upgrade rates, but also have higher rates of rollbacks. In this analysis, we found that only 1/3 of the pull request were merged. We also found that a nearly quarter of the builds fail. Further, developers report several design issues that can cause developers make using automated pull requests less effective. As a result, developers may perform more updates, but several factors can contribute to lower productivity.

D. RQ4: How Do Developers Perceive Out-Of-Date Dependencies and Make Dependency Management Decisions?

Developers were clear that dependency management remained a difficult problem. As one developer responded, "*it is one of the most significantly painful problems with development*". The top three concerns for developers were *breaking changes* (changes that contain syntactical or semantic incompatibilities), *understanding the implications of changes*, and *migration effort*. Surprisingly, *monitoring changes* (the effort associated with checking the availability of new versions of dependencies) was the lowest rated concern. Interestingly, developers closely rated the concerns of losing features and security, suggesting there several developers value preserving features as strongly as staying secure.

1) Awareness Preferences: We asked developer to describe their preferences for how they wished to be informed about changes in dependencies. 54% preferred a passive style notification such as a dashboard or badge, while 46% preferred automated mechanisms.

2) Control: A major theme that emerged from the survey results was the concept of *control*. Several developers rejected the notion of even using package managers, wanting to explicitly manage the process themselves: "*[I am not a] *mindless developer*, who turns that responsibility over to package managers*". Another developer stated:

> *I want the thought of updating to be a deliberate act, not just to get rid of a badge count or something - badge counts *nag* you...*

3) Update Practices: Developers identified several strategies for updating dependencies. These strategies roughly corresponded to a *quick*, *scheduled*, and *reactive* philosophy.

Developers who selected a quick strategy, believed that responding to updates immediately and merging them allows them to incrementally offset the technical debt associated with changing dependencies.

Other developers maintained a scheduled strategy. These developers used a systematic and scheduled method for reviewing changes in dependencies. For example, one developer describes how they reviewed dependencies quarterly, and then recommended upgrades to management. Another developer described how they had a team whose responsibility was to change versions of libraries and packages for the company.

A large set of developers used a reactive strategy. They waited until a situation required that them to update a dependency, i.e., "the squeaky wheel gets the grease". When dealing with multiple teams, this could get more complicated. In one company, updates happen when enough momentum builds up for one team to move to a new version of a library. The team will just perform the update and let the other teams know to work with the new dependencies when they merge.

Finally, one developer cautioned others about using a quick strategy:

> *There is something seductive about being up to date with all your libraries, but the reality is that updating usually costs more than it benefits, so the best strategy is to update infrequently. Sad if you're a library developer, but that's the reality.*

V. Discussion

We discuss several observations related to our findings and provide advice for the design of other automated pull requests.

A. Upgrade Suspicion

Developers are strongly influenced by even a single negative experience in how it shapes their beliefs about software engineering practices [19]. Devanbu and colleagues finds that these beliefs can be difficult to change without being provided with strong arguments. For example, one developer describes their conservative stance on updating a dependency, which is driven by a single bad experience:

> *I've had a large .net control library get hit with a couple of nasty regression bugs after updating to get other bugs fixed. If everything is working now, why would I touch it?*

Even when repositories use continuous integration with extensive test suites, dependency updates can still break. A common reason for this is that the update changes a feature that is difficult to unit test, lacks test coverage, or is a little used feature within the program.

> *Going from nodemailer 0.5.x to 2.3.x exposed a difference in how services are supported. The earlier versions provided defaults for gmail, yahoo, icloud, send grid etc. These have been extracted out into plugins. Not sure we are going to handle this but email isn't a feature that is used a great deal so I've upgraded to the latest that doesn't cause the breaking change: 0.7.1.*

Each automated pull request that a developer receives asks them to confront this *upgrade suspicion*—regardless of how harmless the update maybe. For many developers, this upgrade suspicion will persist. Automated pull requests need to include several measures to counter this effect, including providing arguments for why a developer should consider an upgrade: As one developer states, "*it would take hell of a good reason to even consider it*".

B. Notification Fatigue and Awareness

Developers were mixed in their preference for notifications. While developers appreciated the value of receiving a timely notification, they were concerned with the consequences of yet another source of notifications. They believed any attempt to use a tool that provided version update notifications would either be ignored, become annoying, and overall result in "too many darn notifications".

Badges serve as an interesting alternative to techniques that involve notifications. They are a simple mechanism without any automation support, yet evidence found they could be effective in encouraging developers to update software dependencies. Instead of relying on an automated mechanism to detect that a new version could be updated, badges rely on *social mechanisms*. Badges can introduce social pressure through public shaming to encourage developers to update software dependencies. Other developers can judge when an out-of-date dependency is worth the attention of the project and create a corresponding issue or pull request. In our manual inspection of repositories, we have seen evidence of this practice. Overall, badges offer a relatively effective alternative to "pushy" automated mechanisms.

However, one developer was concerned with the mindset that is created by these tools:

> *[These tools] are annoying. They create an artificial sense of urgency in less experienced and less thoughtful engineers who feel compelled to "keep things green", making it easy to fall into the trap of perpetually responding to changes in dependencies rather than building new value into your product.*

C. Implications for Automation Tool Design

Acharya and colleagues [20] describe a vision where *code drones* autonomously perform simple chores, such as upgrading software dependencies, performance optimizations, or performing simple refactorings. Automated pull requests are a stepping stone to this future, which still allows a human to make a final judgment about a change to the software. The widespread adoption of tools such as greenkeeper suggests that such a future may be here sooner than we expect.

However, the points of friction identified in this work can help inform the design of automation tools. We provide three implications for design that can help inspire how automated pull requests are built:

- **Argumentation.** An agent of automation needs the ability to provide an argument that explains why a change should be made to a software system. This should be supported by evidence and data automatically obtained by the agent (e.g., evidence of sizable improvement to system performance after internal testing of a change). When possible, changes should be supported by empirical evidence of success: *Other updates saw similar levels of performance gains and were not rolled back.*
- **Transparency.** Norman argues that automated systems always need the ability for humans to intervene and takeover an automated process [21]. Unfortunately, he describes how automation in situations such as automated flight has the tendency to work for the easiest tasks and fail for the hardest tasks. To counter this, Norman argues that automated systems need to continuously expose its state and provide the ability for a human to take over at

critical moments. For automated pull requests, it may not be possible to propose a fully functional change. Instead, developers may need to take over and finish the last set of changes in order to use the recommendation. For example, we observed that developers merged code that *did not* build in order to manually make the corrections themselves.

- **Confidence.** Automated pull requests should also take into account confidence measures that developers can use to estimate the risk of performing an update. For example, sometimes merged pull requests are quickly downgraded. If these rollbacks are automatically tracked, then there several opportunities to use this data to calculate a confidence score for upgrade success. Another approach would be to perform A/B testing with developers that are willing to try new updates at a faster cadence, calculate rollback rates, and then roll out updates to other repositories.
  Another tactic can use API method coverage in the client test suite when calculating a confidence score. For example, if a programmer is using the mongodb `sort` method in their program, but lacks tests for that code, then any upgrade that involves a change to the underlying mongodb `sort` method implementation can be considered *risky*. But if the developer adds tests covering usage of the `sort` method, then any upgrade that involves a change to the mongodb `sort` method implementation can now be considered *less risky*. Finally, it could be possible to integrate program analysis techniques which try to predict incompatibilities between different versions of dependencies and a target system [22]. Using all of these measures, a risk index can be calculated for upgrades between pairs of dependency versions.

VI. Related Work

Research has investigated api evolution, software ecosystems, which often include understanding the interconnected nature of dependencies and projects, awareness, and explored techniques for automatic API migration.

A. API Evolution

McDonnell and colleagues [23] conducted an in-depth case study of the Android API and client mobile applications. In the study, they find that client applications typically take 14 months before upgrading to a new version, despite new versions being released every three months. Meanwhile, the API itself changes considerably, causing 28% of API references in a client application to become obsolete. Sawant and colleagues [24] studied five Java APIs and analyzed how their clients reacted to deprecation of API methods. They found that few API clients update their API versions. Unfortunately, the most common reaction to a deprecation is to either ignore or delete the element without replacing it with its recommended counterpart.

Dig and Johnson find in a study of five APIs that structural refactoring of the methods and classes caused most breaking changes [25]. Ruiz and colleagues [7] have investigated the presence of behavioral breaking changes that cannot be detected by signature changes of the API alone. In the study, the authors used the API clients test suite on 68 versions from 15 popular Java software libraries in order to determine if there were any differences in behavior between API versions. As a result, most of API versions (52) contained behavioral breaking changes. Further, these breaking changes could be linked to 144 real world bugs. These observations are consistent with our developer's biggest concern of breaking changes.

These studies provide empirical backing for developers' concerns related to breaking changes and efforts related to migrating code to new versions. In our study, our empirical evidence can provide guidance for social and technical factors relevant to applying these methods.

B. Dependencies and Software Ecosystems

German et al. [2] describe multiple problems associated with managing and specifying dependencies, including downloading, building, and satisfying interdependent artifacts, which may not always be explicitly documented. They propose a framework for categorizing dependency types and a method for building and visualizing an inter-dependency graph of a package. Lungu and colleagues [26] note that dependencies also exist between projects in a software ecosystem. They propose a model, which can capture inter-project dependencies.

Because of these inter-project dependencies, software projects must adopt strategies, that help coordinate how downstream and upstream project changes are handled. Bogart and colleagues [14] studied the Eclipse, R/CRAN, and Node.js/npm ecosystems and they found that each ecosystem used a different strategy which reflected the community's values. For example, the R/CRAN community strongly preferred simultaneous upgrades to the latest versions of dependencies across projects. To understand contextual factors related to managing inter-project dependencies, Bavota and colleagues [27] analyzed dependency changes, mailing lists, and issue tracking systems in 147 Java projects from the Apache community. They found that a client project tends to upgrade a dependency only when substantial changes in the library project are released, including bug-fixes. It is interesting to contrast this with the *quick* style of upgrade management used by some of the developers in our study, which can be facilitated by using automated pull requests.

In a software ecosystems, changes ripple through the entire ecosystem, which can impact multiple parties. We observed this factor when developers described the impact of receiving many pull requests as a consequence from rapid publishing of npm packages. Tools like greenkeeper.io, for better or worse, can amplify the impact of a single author publishing a change across an entire ecosystem.

C. Awareness

Several tools have been developed to help developers maintain awareness of changes in APIs and dependencies. Cadariu and colleagues [3] created a Vulnerability Alert Service (VAS) that can scan Maven dependencies of a project for security vulnerabilities. The scanner checked for Common Vulnerabilities and Exposures (CVE) that existed in the dependency changelogs or code history and automatically generated *alerts* for developers. Apiwave [28] is a tool that tracks API popularity and migration between frameworks and libraries. The tool provides a dashboard visualization and search tool for 320K APIs and tracked evolution of 650 Java projects. While these tools provide the capability to notify and inform developers about changes in APIs, there are several limitations. VAS can alert developers of a security vulnerability, but they cannot provide a recommendation for fixing it. Further, many package management systems, such as npm, already provide this capability directly into the package management system. Apiwave, allows developers to understand changes in popularity of APIs; however, they do not provide usage or migration information related to changes in versions.

Kula and colleagues [1] examined whether security advisories had any influence over library dependency updates. After studying over 4,600 GitHub software projects and 2,700 library dependencies, the researchers found that 81.5% of the studied systems still keep their outdated dependencies and were not likely to respond to a security advisory. Surveys with developers revealed that 69% of the interviewees were simply unaware of the advisory. These results highlight that ineffective methods for maintaining awareness can be a limiting factor, and may explain why other mechanisms, such as badges, can be more engaging.

D. Automatic API Migration

Several researchers have developed techniques for supporting developers in automatically migrating API versions. Many techniques have proposed instrumenting [29], [30] developers' programming environments for tracking the fine-grain evolution of APIs. Other techniques detect differences between API signatures [31] in order to recommend migration scripts for clients. Finally, Nguyen and colleagues [13] have proposed mining client changes in order to infer API migration scripts. Unfortunately, Cossette and Walker [32] caution researchers about the practical effectiveness of these approaches: in their study, no one technique could achieve a fix rate over 20%.

In organizations, such as Google, where all dependencies, toolchains, and projects are checked into a single code repository, other opportunities exist [33]. For example, Rosie enables company-wide migration of APIs by performing a system wide migration across all callsites. These changes are then staged, tested, and sent for code review. Once all changes have been approved, they are instantly merged into HEAD. Replicating this process in open source communities is another matter [14].

The emergence of automated pull requests offers an exciting opportunity for researchers to transfer their techniques into practice. Our findings provide further evidence motivating the concern developers have in migration effort. Further, the fact that over a quarter of automated pull requests resulted in build failures, provides empirical evidence for the need of automated migration support. However, when asked about automatic migration, developers remain deeply skeptical—future work will need to overcome both the technical and social challenges associated with automated API migration.

VII. Limitations

Generalizability. We caution readers to not overgeneralize our results. While we analyzed a large sample of open-source repositories, these results may not extend to proprietary systems, which may operate under different constraints. Further, we examine npm packages, which can include many new and evolving packages. The dynamic nature of Javascript can result in harder to detect breaking changes, which may be less of concern in other languages.

Causality and Project Selection. One threat is related to causality. Our evidence may only suggest correlations between tool use and improved update behavior. There are other possible explanations that can explain our results. For example, one possible explanation is that more experienced developers choose tools such as greenkeeper.io; and as a result, the experienced developers also upgrade dependencies more often. Because we find only correlations, this limits are ability to claim that using a particular tool was certainly the reason why a developer choose to upgrade more often. Another related threat is that by looking for projects that use a particular tool, we could be selecting projects that are more active than control projects. We believe that by comparing two different tools with similar activity levels, we can provide a more fair baseline for comparing automated pull requests. While this somewhat mitigates the threats, this does not eliminate them. Instead, we believe this offers preliminary evidence that can be further validated in future studies.

VIII. Conclusion

Developers need incentives and automated mechanisms that can improve their completion of important maintenance tasks without being too distracting or time-consuming. In this study, we evaluated two mechanisms, (1) automated pull requests, and (2) badges, in order to see if they can encourage developers to upgrade software dependencies. We compared these mechanisms to a baseline set of projects that did not use any mechanism. We also surveyed developers about their experiences and concerns with dependency management and preferences for tool support. Our findings show that automated pull requests can encourage developers to update dependencies quicker and at a higher rate than badges or our baseline. Further, projects with automated pull requests more often have higher versions of the dependencies. Unfortunately, several problems exist with automated pull requests. Higher rates of rollbacks, notification fatigue, gaps in continuous integration, and tool design issues can interfere with a developer's productivity. Badges can provide some benefits over our baseline while reducing these negative aspects of automated pull requests. For researchers, many research challenges remain for improving the user experience of automated pull requests.

Acknowledgement

This material is based in part upon work supported by the National Science Foundation under grant number 1559593.

References

[1] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, May 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[2] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "A model to understand the building and running inter-dependencies of software," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 140–149.

[3] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 516–519.

[4] P. Xia, M. Matsushita, N. Yoshida, and K. Inoue, "Studying reuse of out-dated third-party code in open source projects," *Information and Media Technologies*, vol. 9, no. 2, pp. 155–161, 2014.

[5] O. Foundation, "Top 10 Security Risks," https://www.owasp.org/index.php/Top102017-A9-UsingComponentswithKnownVulnerabilities, 2017, [Online; accessed 8-May-2017].

[6] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985813

[7] E. Ruiz, S. Mostafa, and X. Wang, "Beyond api signatures: An empirical study on behavioral backward incompatibilities of java software libraries," Department of Computer Science, University of Texas at San Antonio, Tech. Rep., 2015. [Online]. Available: http://xywang.100871.net/TechReport_EmpIncomp.pdf

[8] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 112–121. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486804

[9] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 356–366. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568315

[10] M.-A. Storey and A. Zagalsky, "Disrupting developer productivity one bot at a time," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 928–931. [Online]. Available: http://doi.acm.org/10.1145/2950290.2983989

[11] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on GitHub," in *12th Working Conference on Mining Software Repositories*, ser. MSR. IEEE, 2015, pp. 367–371.

[12] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 345–355. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568260

[13] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *SIGPLAN Not.*, vol. 45, no. 10, pp. 302–321, Oct. 2010. [Online]. Available: http://doi.acm.org/10.1145/1932682.1869486

[14] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950325

[15] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 2–12.

[16] 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. New York, NY, USA: ACM, 2017, pp. 385–395. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106267

[17] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 426–437. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970358

[18] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding In-depth Semistructured Interviews," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, aug 2013. [Online]. Available: http://journals.sagepub.com/doi/10.1177/0049124113500475

[19] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *ICSE '16*, 2016, pp. 108–119. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884812

[20] M. P. Acharya, C. Parnin, N. A. Kraft, A. Dagnino, and X. Qu, "Code drones," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 785–788. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889211

[21] D. A. Norman, "The'problem'with automation: inappropriate feedback and interaction, not'over-automation'," *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, vol. 327, no. 1241, pp. 585–593, 1990.

[22] S. McCamant and M. D. Ernst, "Early identification of incompatibilities in multi-component upgrades," in *European Conference on Object-Oriented Programming*. Springer, 2004, pp. 440–464.

[23] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 70–79. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.18

[24] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular java apis," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 400–410.

[25] D. Dig and R. Johnson, "How do apis evolve&quest; a story of refactoring: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 83–107, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1002/smr.v18:2

[26] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 309–312. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859058

[27] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: An evolutionary study," *Empirical Softw. Engg.*, vol. 20, no. 5, pp. 1275–1317, Oct. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-014-9325-9

[28] A. Hora and M. T. Valente, "Apiwave: Keeping track of api popularity and migration," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 321–323.

[29] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support api evolution," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 274–283. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062512

[30] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 19:1–19:35, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000799.2000805

[31] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818–836, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70747

[32] B. E. Cossette and R. J. Walker, "Seeking the ground truth: A retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 55:1–55:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393661

[33] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Commun. ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2854146