There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm

BENJAMIN ROMBAUT, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada
FILIPE R. COGO, Centre for Software Excellence (CSE) at Huawei, Canada
BRAM ADAMS, Lab on Maintenance, Construction, and Intelligence of Software (MCIS) at Queen's University, Canada
AHMED E. HASSAN, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada

Dependency management bots are increasingly being used to support the software development process, for example to automatically update a dependency when a new version is available. Yet, human intervention is often required to either accept or reject any action or recommendation the bot creates. In this paper, our objective is to study the extent to which dependency management bots create additional, and sometimes unnecessary, work for their users. To accomplish this, we analyze 93,196 issue reports opened by Greenkeeper, a popular dependency management bot used in open source software projects in the npm ecosystem. We find that Greenkeeper is responsible for half of all issues reported in client projects, inducing a significant amount of overhead that must be addressed by clients, since many of these issues were created as a result of Greenkeeper taking incorrect action on a dependency update (i.e., false alarms). Reverting a broken dependency update to an older version, which is a potential solution that requires the least overhead and is automatically attempted by Greenkeeper, turns out to not be an effective mechanism. Finally, we observe that 56% of the commits referenced by Greenkeeper issue reports only change the client's dependency specification file to resolve the issue. Based on our findings, we argue that dependency management bots should (i) be configurable to allow clients to reduce the amount of generated activity by the bots, (ii) take into consideration more sources of information than only the pass/fail status of the client's build pipeline to help eliminate false alarms, and (iii) provide more effective incentives to encourage clients to resolve dependency issues.

CCS Concepts: • **Software and its engineering → Software design engineering**; **Software design tradeoffs**; **Software maintenance tools**.

Additional Key Words and Phrases: Dependency Management, Software Bots, Mining Software Repositories, Greenkeeper, Overhead

ACM Reference Format:
Benjamin Rombaut, Filipe R. Cogo, Bram Adams, and Ahmed E. Hassan. 2022. There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2022), 40 pages. https://doi.org/10.1145/3522587

Authors' addresses: Benjamin Rombaut, benjamin.rombaut@queensu.ca, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Kingston, Canada; Filipe R. Cogo, filipe.roseiro.cogo1@huawei.com, Centre for Software Excellence (CSE) at Huawei, Kingston, Canada; Bram Adams, bram.adams@queensu.ca, Lab on Maintenance, Construction, and Intelligence of Software (MCIS) at Queen's University, Kingston, Canada; Ahmed E. Hassan, ahmed@cs.queensu.ca, Software Analysis and Intelligence Lab (SAIL) at Queen's University, Kingston, Canada.

1 INTRODUCTION

Today's software systems are rarely built from scratch, with client projects often making use of specific versions of provider packages (libraries) in the form of dependency relationships. Because dependency relationships enable code reuse, they have been shown to improve developer productivity, software quality, and time-to-market of software products [1, 2]. However, this often comes at an increased cost on the client's part of having to actively manage their dependencies [38], as provider packages continuously release new versions containing bug fixes, new functionalities, and security enhancements, and these updates can break API-backwards compatibility [7].

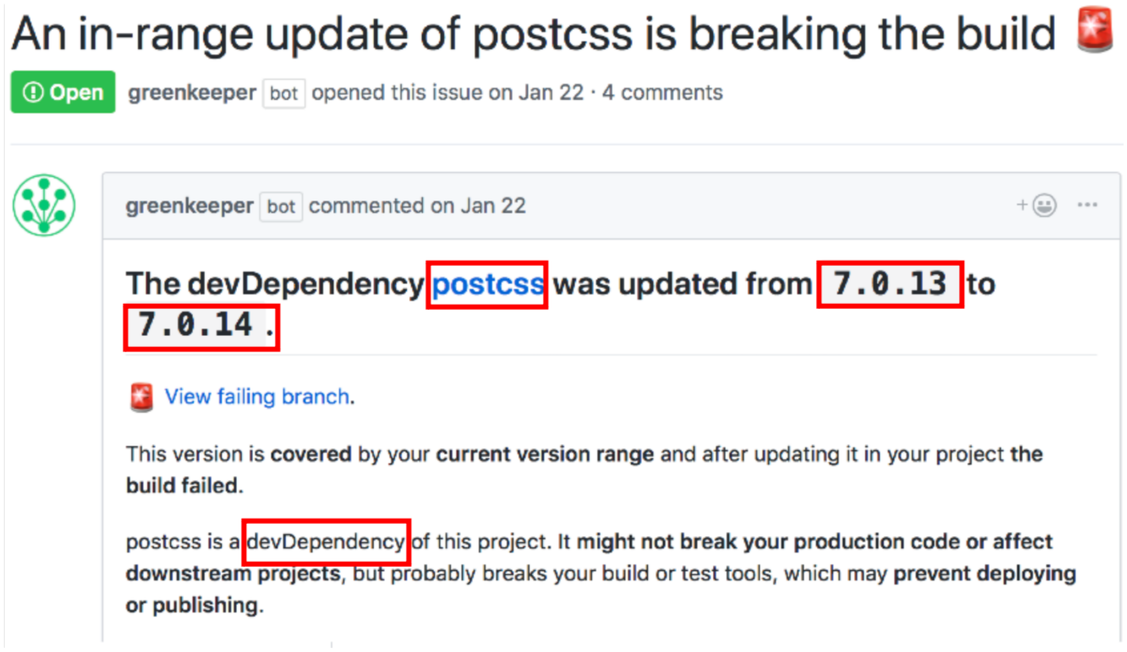
Clients are increasingly turning to software bots to alleviate the cost of managing their dependencies. The aim of these bots is to reduce the workload of repetitive tasks faced by practitioners (e.g., updating the client's dependency constraints when a provider releases a new version) and to notify client packages about dependency updates that break their build (e.g., automatically testing new dependency releases that satisfy the client's accepted dependency version range). In fact, studies have shown that dependency management bots can help with encouraging developers to keep their dependencies up-to-date [3, 38], and detecting and reporting build failures [34, 50, 52].

Integrating these bots into a project's workflow requires a certain level of effort on the part of the client developers, and once the bot begins performing its specific function, human intervention is usually required to either accept or reject any action or recommendation the bot creates. One such dependency management bot that clients can integrate into their projects is Greenkeeper[1]. Each time one of the providers a client depends on releases a new version, Greenkeeper opens a new branch in the client project with that update. The client's continuous integration (CI) tests kick in, and Greenkeeper watches them to see whether they pass or fail. If the client's CI pipeline fails with the new provider version and the provider release is within the accepted dependency version constraints specified by the client, the bot will create a Greenkeeper issue report (GKIR) in the client's repository with information stating which dependency caused the issue. Figure 1 provides an example of a GKIR with the provider package name, current version, target version, and the dependency type in the client project highlighted.

Regular users of the client package could potentially be affected by these GKIRs, so there is incentive for clients to resolve GKIRs in a timely manner, yet this is not always possible in an automated way. For example, if it is discovered that a new release of the provider is breaking the client, dependency management bots often recommend downgrading a dependency to an older version. This downgrade occurs by modifying the client's dependency version constraints to only accept a specific older version (a.k.a., *version pinning*). While this has been shown to be one of the most applied workarounds requiring the least effort to resolve dependency issues [28], it introduces a host of other issues that can affect the client unless manual measures are taken to constantly update the dependency constraint to a newer version. For example, older versions of a provider can contain security issues, and more recent versions of providers often include fixes related to project stability [15]. In other words, tools like Greenkeeper will automatically attempt to version pin the offending dependency when a GKIR is initially opened, in effect recommending to the clients to employ an anti-pattern in their project.

While previous studies have found that bots are able to automate dependency updates [38, 52], there is a lack of research investigating the introduced overhead that accompanies integrating with these bots, the efficacy of common actions recommended by bots for resolving dependency issues, or the size of the changes that are required to be made by client developers to resolve the issues reported by these bots. Therefore, in this paper we perform an empirical study of four years of Greenkeeper data to examine the extent to which automated dependency management bots can

[1]https://greenkeeper.io/


An in-range update of postcss is breaking the build

Open greenkeeper bot opened this issue on Jan 22 · 4 comments

greenkeeper bot commented on Jan 22

The devDependency postcss was updated from 7.0.13 to 7.0.14.

View failing branch.

This version is **covered** by your **current version range** and after updating it in your project **the build failed.**

postcss is a devDependency of this project. It **might not break your production code or affect downstream projects**, but probably breaks your build or test tools, which may **prevent deploying or publishing.**

Fig. 1. An example of a Greenkeeper in-range breaking build update issue report with the provider package name, current version, target version, and the dependency type in the client project highlighted.

either save or create unnecessary work in their client projects, and report on the lessons learned from our analysis. Specifically, we investigate the following research questions:

RQ1: What is the overhead introduced in client projects by Greenkeeper? We observe that Greenkeeper generates a significant amount of artifacts (e.g., issue reports (IRs) and comments) that must be addressed by clients. GKIRs make up approximately half of the IRs in all projects, or two-fifths in projects with a high number of issue reports. The Greenkeeper bot itself generates nearly all of the activity on GKIRs. The vast majority of comments on GKIRs are from Greenkeeper, with more comments continuing to be generated the longer the GKIR remains open. Approximately one-fifth of user comments on GKIRs indicate that the GKIR is a *false alarm*, meaning that, while the client's CI pipeline might have failed with the new provider release applied, the CI failure was not in fact caused by the new provider release, and that the GKIR only serves to create noise in the client project.

RQ2: Is automated dependency pinning an effective mechanism for resolving GKIRs? To our surprise, we observe that automatically attempting to pin the dependency turns out to be a relatively ineffective solution to resolving GKIRs, failing over two-thirds of the time. Yet, since the updated dependency is the only difference between the GKIR branch and the project's main branch, pinning the dependency to the previous version (i.e. the version that was previously in use on the project's main branch) should in effect render the GKIR branch a duplicate of the project's main branch. After further manual analysis, we observe that GKIRs with pin attempts that fail are caused by issues unrelated to the dependency being updated, such as misconfigured pipeline environments, and often are in fact false alarms that are unrelated to the dependency update.

RQ3: What are the performed code changes when resolving GKIRs? We observe that more than half (56%) of commits that resolve GKIRs only modify dependency specification files, and that 57% of these commits only modify a single line in the client's dependency specification file, usually to upgrade the dependency version specification. Commits referenced by GKIRs that require changes to the client's code are comparable in size to commits referenced by non-GKIRs, and tend to include changes to a mixture of different file types.

The aforementioned findings show that significant overhead can be introduced in client projects by dependency management bots in the form of numerous notifications and false alarm issues. To reduce this overhead, we argue that dependency management bots should take into account more fine-grained information than simply the pass or fail status of the client's CI pipeline when attempting to update a dependency. Specifically, bots should be able to distinguish between CI pipeline failures caused by existing issues in the client's project (e.g., incompatible Node version errors) and valid CI pipeline failures caused by the updated dependency. Additionally, dependency management bots should be mindful of the trade-offs introduced by different features that could increase or reduce the overhead introduced by the bot.

Our study presents the following contributions:

- An empirical investigation of the overhead introduced by dependency management bots (RQ1), the efficacy of recommended actions by the bot (RQ2), and the size of manual changes required by developers to resolve issues created by the bot (RQ3);
- A discussion of practical implications for designers of automated dependency management bots;
- A dataset to help foment further empirical investigations on the related fields. In addition, we make our parsers used to extract dependency information from GKIRs public, so that they can be reused by developers and researchers to aid further studies [48].

2 BACKGROUND & RELATED WORK

In this section, we provide a more in-depth description and discuss the existing work concerning dependency management and software bots.

2.1 Dependency Management

As most client packages use provider packages through dependency relationships [23, 54], it is important to standardize how these relationships are tracked. Semantic Versioning[2] (SemVer) has become a popular policy for communicating the kinds of changes made to a software package. A SemVer-compatible version is a version number composed of a major, minor and patch number that allows maintainers to logically order package releases. The efficacy of this policy has been studied in previous work [16, 19, 45, 46].

Amongst others, SemVer is adopted in the npm[3] ecosystem[4], where provider packages need to specify the version number of each release in their *package.json* metadata file[5]. In turn, client packages can designate a dependency relationship with a provider package in their *package.json* file as either a runtime dependency, which is required by the client package in a production environment, or a development dependency, which is only needed by the client package for local development and testing.

[2]https://semver.org
[3]https://www.npmjs.com/
[4]https://docs.npmjs.com/misc/semver
[5]https://docs.npmjs.com/files/package.json

In addition to the dependency relationship type, clients can specify whether they would like to accept either a specific version or a range of versions from the provider. If a specific version is used (i.e., *version pinning*), the client will only accept that unique version of the provider. If a version range is used, the provider is implicitly updated whenever a new release from the provider is available that satisfies the existing version range statement in the client package (i.e., *in-range update*). Version ranges are constrained using a set of operators that specify versions that satisfy the range (e.g., "^" to accept only *minor* and *patch* updates, "~" to accept only *patch* updates, etc.). For example, if a client specifies an accepted version range of `^1.0.0` for a provider, and that provider releases version `1.0.1`, that provider update is in-range for the client, and will be implicitly updated. In other words, if other developers clone the client project (i.e., client developers) and install the client's dependencies, they would receive version `1.0.1` of the provider package. Additionally, if other projects make use of the client's package as a dependency (i.e., client users) and therefore transitively depend on the client's dependencies, they would also receive version `1.0.1` of the provider package when they install their dependencies, which includes the client's published package.

To illustrate this, Figure 2 shows an example of how the dependency versioning statement of a client package C for a provider package P affects the resolved version of P that is used by C when C is built. Initially at T_0, P has released version `1.0.0` and C specifies a versioning constraint as the range statement `"P": "^1.0.0"` (i.e., implicitly accepting versions ≥1.0.0 ∧ <2.0.0 of P). Therefore, when C is built, it will use version 1.0.0 of P. At T_1, P releases version `1.0.1`. Because this version falls within C's accepted version range, this version will now implicitly be used when C is built. At T_2, C changes their dependency version statement for P from `"P": "^1.0.0"` to `"P": "1.0.0"` (i.e., pinning P to version `1.0.0`). Now when C is built, only version `1.0.0` of P will be used. This can be seen at T_3, where P releases version `1.0.2`, but C will continue to explicitly use version `1.0.0`. In other words, not only will C no longer benefit from any new features released by P in the future, but they have also implicitly downgraded P from `1.0.1` to `1.0.0`. At T_4, C again decides to modify their dependency version statement, this time changing from `"P": "1.0.0"` to `"P": "^1.0.0"` (i.e., unpinning P and again implicitly accepting versions ≥1.0.0 ∧ <2.0.0 of P). Now when C is built, version `1.0.2` of P will be used. Finally, at time T_5, P releases version `1.0.3`, which falls within C's accepted version range, and therefore will now be used when C is built.

While practitioners stand to reap the many benefits that come with reusing software systems that have been previously built and are maintained by other developers, these dependencies often come with the increased cost of having to be managed and updated. An important decision that is faced by clients is whether they should constrain a dependency to a single specific version, or automatically accept a range of versions from the dependency. By constraining their dependencies, clients are able to drastically reduce the risk of a dependency update breaking their project. In fact, Bogart et al. [6] found the risk of breaking updates to be one of the main concerns that clients have when determining whether they should update their dependencies. However, clients must manually modify their dependency constraints if they want to take advantage of bug fixes and new features in specific versions as they are released by the provider.

By accepting a range of versions from a dependency, clients are able to automatically receive minor updates as they are released, potentially reducing the overhead of managing their dependencies. However, there is a risk that providers will not respect the SemVer policy [10, 25, 29] and release new versions that are not backwards-compatible. Situations where the new provider release falls within the client's range of accepted versions and ends up breaking the client's build (i.e., an *in-range breaking update*) can be a major problem for clients, especially if the provider is a runtime dependency, as both client developers and client users will be impacted while the issue remains unaddressed, being unable to successfully build or install the client project.

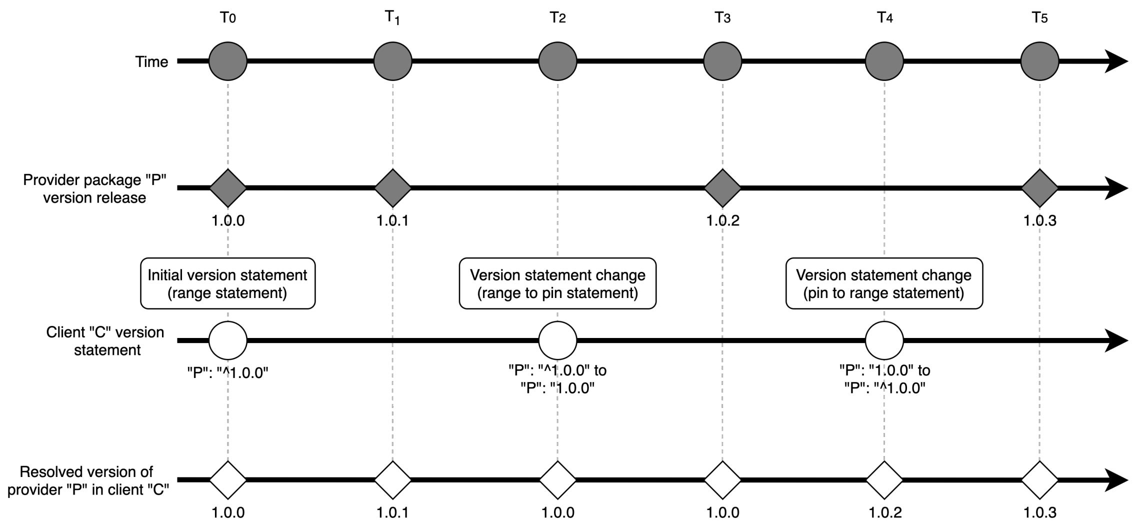

T0
T1
T2
T3
T4
T5
Time
Provider package "P" version release
1.0.0
1.0.1
1.0.2
1.0.3
Initial version statement (range statement)
Version statement change (range to pin statement)
Version statement change (pin to range statement)
Client "C" version statement
"P": "^1.0.0"
"P": "^1.0.0" to "P": "1.0.0"
"P": "1.0.0" to "P": "^1.0.0"
Resolved version of provider "P" in client "C"
1.0.0
1.0.1
1.0.0
1.0.0
1.0.2
1.0.3

Fig. 2. An example of how the dependency versioning statement of a client package *C* for a provider package *P* affects the resolved version of *P* that is used by *C* when *C* is built.

Client developers can protect themselves from in-range breaking updates by using *lock files* (e.g., `package-lock.json`) in their project. The lock file will describe the entire dependency tree of the project as it is resolved when created, including nested dependencies with specific versions. The lock file is intended to pin down (i.e., lock) all versions for the entire dependency tree at the time that the lock file is created, and is usually included to the client projects repository, so that other client developers can install the exact dependencies specified in the lock file. In other words, this ensures that installations remain identical and reproducible throughout the client project's entire dependency tree, across other developers, such as team members working together, and across systems, such as when running a CI build.[6]

However, while including a lock file in the client's repository might protect other client developers from in-range breaking updates, it does not protect the users of the client package from these issues. This is because 'package-lock.json' cannot be published to `npm`[7]. This means that if a user of the client package (e.g., another developer with their own project) installs the client's published package from `npm` (rather than, say, another client developer cloning the git repository), the user will never download the client's `package-lock.json` file, and therefore it will be completely ignored when the client's dependencies (transitive dependencies to the user of the client package) during the installation, allowing users of the client's package to use any version of the client packages dependencies that are compatible with the version ranges dictated by the client's `package.json` file. This is done by `npm` in order to reduce the amount of package duplication caused when lots of a package's dependencies all depend on slightly different versions of the same transitive dependency.

[6]https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json
[7]https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json#package-lockjson-vs-npm-shrinkwrapjson

Multiple studies have examined how to detect breaking changes in API updates [9, 29, 33]. Specifically, Mezzetti et al. [37] and Møller and Torp [42] describe the NoRegrets and NoRegrets+ tools, respectively, that generate models for both the pre-update and the post-update version of a provider, then compare the models to identify type regressions. Mujahid et al. [39] describe a crowd-based approach for detecting breaking changes in provider releases by leveraging the automated test suites of multiple client projects that depend upon the same dependency to test newly released versions.

2.2 Recovering from Dependency Issues

If a dependency update breaks a client's build, the client may resort to version pinning their dependencies to resolve the issue [13, 30]. Version pinning a dependency involves changing the dependency specification from a range statement to a specific version statement, in effect locking the dependency to the previously known working release, as can be seen at T_2 in Figure 2. In this example, the provider P may have made a backwards-incompatible change when they released version `1.0.1`, creating an issue in the client package C and prompting C to pin P to the previous working version (i.e., `1.0.0`).

Version pinning a dependency is a common practice, usually motivated as a workaround to fix breaking updates that occur from a dependency releasing a backwards-incompatible change. Jafari et al. [28] found that developers choose to pin some of their dependencies in over 52% of npm projects, and Cogo et al. [13] found in their study on dependency downgrades that 49% of all downgrades occur due to a replacement of a version range statement with a specific version (i.e., pinning the dependency).

Pinning is a legitimate option when developers do not have the time or resources to fix an issue introduced by a dependency update, as pinning is the action that requires a minimum overhead to potentially resolve these type of issues. However, unless manual measures are taken to update the dependency constraint when new versions are released, the client will not receive bug fixes or new functionalities from the provider [28]. Additionally, Zerouali et al. [56] found that technical lag, which is used to quantify packages lagging behind with respect to using the latest version of their dependencies, is often caused by clients using strict dependency version constraints.

To detect whether clients would be affected by a dependency update, Møller et al. [41] propose a simple pattern language for describing API access points that are involved in breaking changes, and provide an accompanying program analysis tool for locating which parts of the client code may be affected by the breaking change. Nielsen et al. [44] go a step further with their tool JSFIX, which detects the locations affected by breaking changes in dependency updates, then transforming those parts of the code to become compatible with the new provider version.

2.3 Dependency Bots

Bots can be defined as tools that perform repetitive predefined tasks to save developers' time, increase their productivity, and support them in making smarter decisions [11, 52]. As Storey and Zagalsky [50] and Lin et al. [34] found, bots are useful for automatically completing a wide variety of chores, such as dependency management, detecting flaky tests, and creating issues when a service fails. Wessel et al. [52] showed that bots are primarily used for reporting build failures, decreasing code review time, and automating CI pipelines. Bots are especially prevalent in the area of dependency management [3, 22, 32, 38, 51, 53].

Greenkeeper, Dependabot[8], and Renovate[9] are popular dependency management bots that clients can integrate into their projects to automate dependency updates. All of these bots perform the same overarching task: to help clients keep their dependencies up-to-date. This is accomplished by monitoring the client's dependencies, and automatically testing new dependency releases to see whether they are compatible with the client project. When one of a client's dependencies releases a new version, the bot will create a fresh branch with the new version applied, run the client's CI pipeline, and notify the client of the results with the option to update their dependency specifications. In our study, we focus on data from Greenkeeper because the artifacts Greenkeeper creates (i.e., GKIRs) are easily identifiable and require client developers' attention. Recall that in-range breaking updates can potentially affect the users of a client's project, and therefore should be given special attention by client developers. Greenkeeper also takes care of corner cases that don't require client developer's attention (e.g., when client's pin their dependencies), which makes the data more suitable and reliable to study the phenomenon of interest. Additionally, clients must make the deliberate decision to integrate with Greenkeeper (as opposed to Dependabot, for example, which is automatically enabled on client projects on GitHub to open security PRs[10], which clients may end up paying less attention). These attributes make Greenkeeper an ideal dependency management bot to study in the context of generating overhead for clients. Therefore, although we explain in further detail how Greenkeeper works, the previously mentioned dependency management bots work in a similar manner.

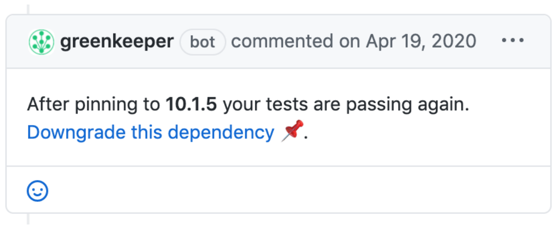
Greenkeeper sits between the client and their ecosystem package manager, watching the providers the client depends on. Each time one of the providers releases a new version, Greenkeeper creates an isolated branch with that dependency update. The repository's CI pipeline runs on the new branch, and Greenkeeper watches the results to see whether they are successful. Based on the test results and the client's dependency constraints, Greenkeeper will open a GKIR in the client's repository with information stating which dependency update caused the problem, an example of which can be seen in Figure 1.

Since dependency updates that cause GKIRs to be created are *in-range breaking updates*, they can directly affect users of the client package if the offending dependency is a runtime dependency. As we discussed in Section 2.1, since the client's dependency constraints automatically allow for the new version of the dependency to be accepted when users install the client package, users will be unable to successfully build or install the client package while the GKIR remains unresolved, even if the client makes use of lock files in their project. Therefore, GKIRs can represent major issues for the client, and there is incentive for the client developers to resolve them in a timely manner.

In addition to being alerted when a new GKIR is created in their project, clients will also receive notifications for any activity that occurs on these GKIRs. One of the biggest drivers of these notifications is Greenkeeper itself. When a GKIR is first created, Greenkeeper will often attempt to pin the dependency that caused the GKIR to be opened, as explained before. Greenkeeper will then comment on the GKIR whether the client's tests are passing again with the pinned dependency. If the client's tests continue to fail, the client must manually resolve the issue with a solution that potentially induces a higher level of overhead. This could include either adapting their codebase to be compatible with the new release of the provider, or by downgrading to an earlier version of the provider that their project is compatible with. However, if the pin is successful, Greenkeeper provides a link to create a pull request (PR) that commits the pinned dependency version specification to the main repository branch. Figure 3 shows an example of Greenkeeper notifying

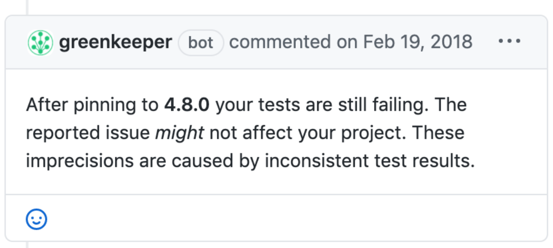
[8]https://github.com/dependabot
[9]https://renovatebot.com
[10]https://docs.github.com/en/code-security/supply-chain-security/managing-vulnerabilities-in-your-projects-dependencies/configuring-dependabot-security-updates#supported-repositories

the client that their tests are passing again (3a) versus that their tests are still failing (3b) with the affected provider version pinned to the previous release.



After pinning to **10.1.5** your tests are passing again. Downgrade this dependency 📌.

(a) A Greenkeeper comment indicating that the client's tests are passing again with the provider pinned to the previous version.

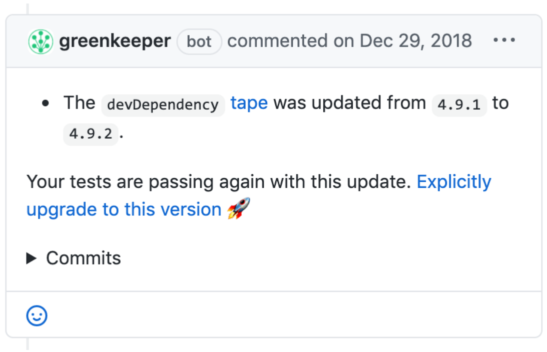


After pinning to **4.8.0** your tests are still failing. The reported issue *might* not affect your project. These imprecisions are caused by inconsistent test results.

(b) A Greenkeeper comment indicating that the client's tests are still failing after pinning the provider to the previous version.

Fig. 3. Examples of comments from Greenkeeper providing the results of automatically attempting to pin the provider when a GKIR is created.

Greenkeeper will continue to generate activity on GKIRs while the GKIR remains open. For example, if the dependency that caused the GKIR to be created releases a new version while the GKIR is still open, Greenkeeper will automatically re-run the client's tests with the new version of the dependency and notify the client whether their tests are passing again with the new version by commenting on the GKIR thread. Figure 4 shows an example of Greenkeeper notifying the client that their tests are passing again (4a) versus that their tests are still failing (4b) with a new version of the provider that caused the GKIR to be created. This feature allows for clients to be notified if the GKIR can actually be resolved by actions taken by the provider, with minimal effort on the client's part. For example, the provider may realize they had released a breaking update, and perform a rapid release to correct the issue. Instead of rushing to fix the GKIR on their side, clients can simply wait for the provider to fix the issue, and Greenkeeper will notify the client if the issue is resolved.

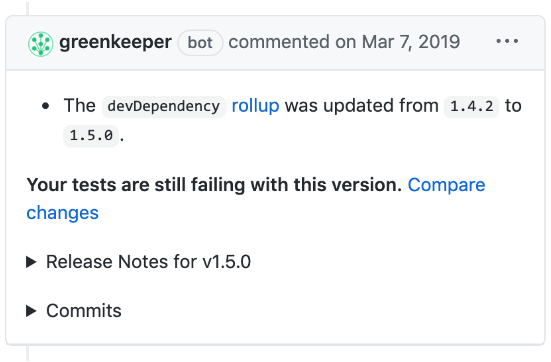


- The `devDependency` tape was updated from `4.9.1` to `4.9.2`.

Your tests are passing again with this update. Explicitly upgrade to this version 🚀

▶ Commits

(a) A Greenkeeper comment indicating that the client's tests are passing again with a new release of the provider tape that caused a GKIR to be created.



- The `devDependency` rollup was updated from `1.4.2` to `1.5.0`.

Your tests are still failing with this version. Compare changes

▶ Release Notes for v1.5.0

▶ Commits

(b) A Greenkeeper comment indicating that the client's tests are still failing with a new release of the provider rollup that caused a GKIR to be created.

Fig. 4. Examples of comments from Greenkeeper when a new version from a provider that caused a GKIR is released.

Greenkeeper recognizes that clients may depend on multiple subpackages from the same provider that are maintained in the same related codebase (i.e., *monorepo package*)[11]. For example, if a client were to depend on the Jest[12] provider, the client could depend on the core jest package, as well as the jest-cli and the jest-resolve sub-packages. These monorepo packages tend to release new versions of their subpackages as a group, and if Greenkeeper were to treat each of these subpackages individually, the clients would be flooded with new notifications for each subpackage they depend on when the provider releases an update. Therefore, in an effort to reduce the overhead introduced to the client, Greenkeeper will group releases from a predefined set of popular monorepo providers together (e.g., Angular[13], Babel[14], Jest, React[15], etc.) into bundled IRs.

Bots and other third-party tools like repository badges that aim to ease the task of dependency management for developers have previously been studied. Alfadel et al. [3] examine the use of Dependabot for automatically creating PRs to fix dependency vulnerabilities in a client's project. They found that approximately 65% of Dependabot security PRs are merged and integrated in the projects, usually within a day of being opened, and that 94% of PRs that are not merged are closed by Dependabot itself. Interestingly, they found that half of the PRs that are not merged were closed by Dependabot because a newer version of the affected dependency was released. Their work specifically examines the efficacy of Dependabot for increasing awareness of dependency vulnerabilities and whether the tool helps developers mitigate vulnerability threats in JavaScript projects,whereas we focus in our study on the potential overhead introduced by dependency management bots.

Mirhosseini and Parnin [38] conducted a study on the effectiveness of different notification techniques designed to help developers keep their dependencies up-to-date. They found that projects that use PR notifications in the form of dependency management bots (e.g., Greenkeeper) and projects that use badge notifications (e.g., David-DM[16]) upgraded their dependencies 1.6 and 1.4 times as often, respectively, as projects that did not use any tools. While their work specifically looks at whether tools like Greenkeeper can help developers keep their dependencies up-to-date, in our study, we look to measure the degree of unnecessary work that these types of tools create in client projects that use them.

3 DATA SET

In this section, we discuss how we collected the data set to address the RQs outlined in the introduction. We use the workflow of Figure 5: (i) we identify projects on GitHub[17] that use the Greenkeeper bot, (ii) we collect all IRs for each project identified in the previous step, and extract the necessary information from each GKIR, (iii) we collect any supporting artifacts related to each GKIR identified in the previous step. Next, we provide a more in-depth explanation of each step in our data-collection workflow.

3.1 Identify projects using Greenkeeper

To identify the projects using Greenkeeper, we first identify projects containing GKIRs. For this, we leverage the title of the GitHub IRs, since GKIRs have a consistent prefix for their titles, namely "*An in-range update of...*", as well as a `user.login` attribute of *greenkeeper[bot]*, and can therefore be distinguished from non-GKIRs. We use the GitHub

[11]https://greenkeeper.io/docs.html#monorepo-dependencies
[12]https://github.com/facebook/jest
[13]https://github.com/angular/angular
[14]https://github.com/babel/babel
[15]https://github.com/facebook/react
[16]https://david-dm.org/
[17]https://github.com/

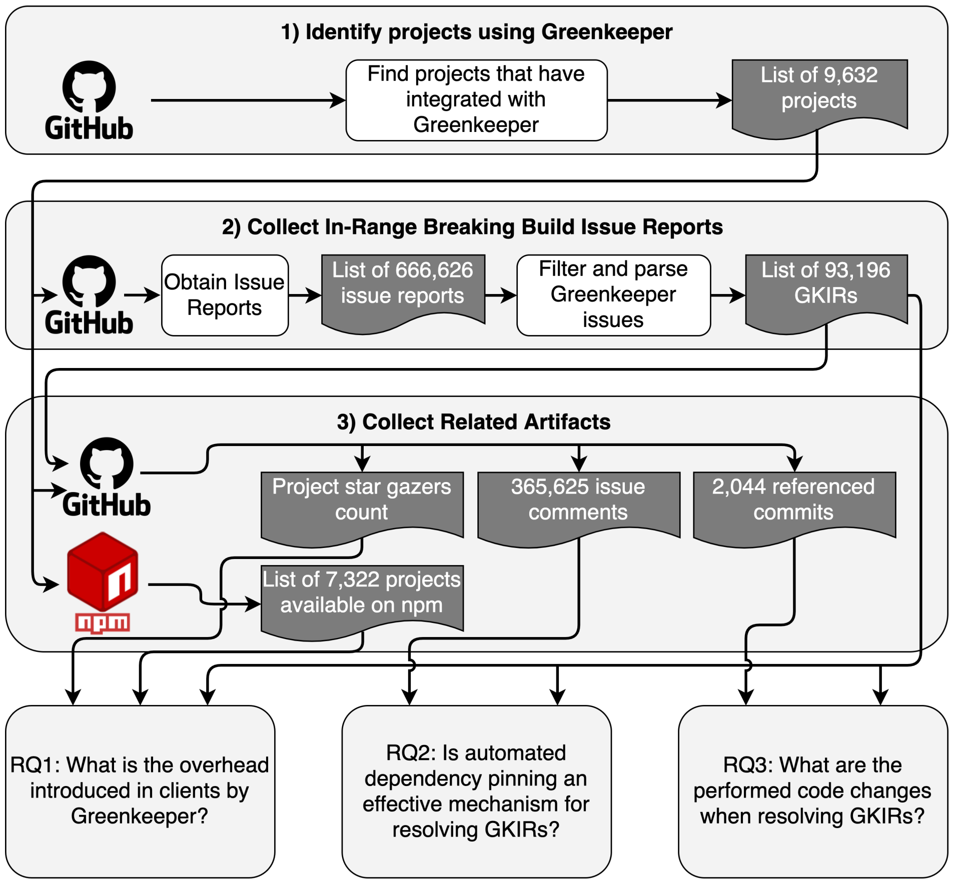

1) Identify projects using Greenkeeper
GitHub
Find projects that have integrated with Greenkeeper
List of 9,632 projects
2) Collect In-Range Breaking Build Issue Reports
GitHub
Obtain Issue Reports
List of 666,626 issue reports
Filter and parse Greenkeeper issues
List of 93,196 GKIRs
3) Collect Related Artifacts
GitHub
Project star gazers count
365,625 issue comments
2,044 referenced commits
npm
List of 7,322 projects available on npm
RQ1: What is the overhead introduced in clients by Greenkeeper?
RQ2: Is automated dependency pinning an effective mechanism for resolving GKIRs?
RQ3: What are the performed code changes when resolving GKIRs?

Fig. 5. Overview of the data collection process.

Search API[18] to search for IRs on GitHub that match this criteria. Each IR record has an associated project attribute. So, once we identify which IRs are GKIRs, we are able to construct a list of GitHub projects that have integrated with the Greenkeeper bot and received at least one GKIR. One of the prerequisites to integrate with Greenkeeper is that the project must have at least one *package.json* file somewhere in the project[19], which implies that all client projects that have integrated with Greenkeeper are part of the npm ecosystem. In total, we extract a list of 9,632 GitHub projects.

3.2 Collect and parse GKIRs

To build our data set of IRs, we use the GitHub API [20] to retrieve all IRs opened in the list of projects we collected in Section 3.1. This step is necessary as a follow up to the step in Section 3.1 to make sure we capture all IRs from these projects, not just GKIRs. We separate the IRs into GKIRs and non-GKIRs using the same criterion described in Section 3.1. GitHub considers PRs to be a type of issue, however we exclude PRs from our analysis, as our study focuses

[18]https://docs.github.com/en/rest/reference/search
[19]https://greenkeeper.io/docs.html#prerequisites
[20]https://docs.github.com/en/rest

on actual IRs, rather than the review process involved with dependency management. Overall, this process leaves us with 93,196 GKIRs and 573,430 non-GKIRs.

To understand the types of providers and provider updates that cause GKIRs, we extract the name of the provider package, the current version of the provider used by the client and the newly released target version of the provider, and whether the provider is a development dependency or a runtime dependency in the client project. For the majority of GKIRs, this information is included in the body of the GKIR. However, in some cases, depending on the version of Greenkeeper used in the client project, not all of this information is available on the GKIR. Specifically, we are not able to extract the provider dependency type from 4% of GKIRs. When this information is required for our analysis, we omit the GKIRs that are missing this data from our study.

The format of the GKIRs is not always consistent. For example, Greenkeeper will group releases from a predefined set of popular monorepo providers together (e.g., Angular, Babel, Jest, React, etc.). Additionally, clients can manually specify whether certain provider releases can be grouped together for their projects. This means that all provider updates made by Greenkeeper will be bundled together into a single GKIR[21], with information about each provider in the bundle in the body of the same GKIR. We found that overall 4.3% of GKIRs correspond to bundled GKIRs. We identify 9 unique GKIR templates based on the version of Greenkeeper that the client was using at the time the GKIR was created, as well as whether the GKIR contained bundled updates. To parse each of these templates, we build 9 unique parser implementations that are able to detect the type of GKIR and extract the necessary information from the GKIRs using regular expressions. We make our parsers available for reuse by developers and researchers, as well as to verify the parsers' correctness [48].

3.3 Collect artifacts related to GKIRs

To understand the activity generated on GKIRs and the maintenance level required to resolve GKIRs, we gather any artifacts related to the GKIRs collected in the process described in Section 3.2. In particular, we retrieve all comments on each GKIR, as well as any commits referenced by GKIRs to analyze the level of activity generated on GKIRs and the types of changes developers create to resolve GKIRs, respectively. These artifacts are retrieved in the form of issue events[22], which are created whenever an interaction related to the issue occurs (e.g., a user references the IR from a PR). We use the `user_login` attribute on the GitHub comment records to distinguish between comments left by users and comments left by bots, collecting a total of 10,724 comments from users on GKIRs, 354,901 comments from bots on GKIRs, and a total of 2,044 unique commits referenced by GKIRs.

Finally, we collect the number of stargazers each project has on GitHub, which we use as a measure of popularity of the project [8]. We also determine whether the project is available as a provider package by searching for the project's name on the npm registry. In order for a project to be available to download in the npm ecosystem as a provider package, it must be available on the npm registry. However, a client package can make use of a provider package available on the npm registry without itself being available on the registry (i.e., it is possible for a project to act as a client, a provider, or both on npm). We find that 76.1% (7,322) of the projects in our dataset are available to download on the registry. We use these projects specifically to explore how long it takes for developers to address GKIRs, since GKIRs can potentially affect the users of these packages.

The collected data is available as part of our supplementary package [48].

[21] https://greenkeeper.io/docs.html#monorepo
[22] https://docs.github.com/en/rest/reference/issues#events

4 RESULTS

In this section, we present the results for each of our RQs. For each RQ, we discuss the motivation, the approach we used to address the RQ, and our findings.

4.1 RQ1: What is the overhead introduced in client projects by Greenkeeper?

Motivation. While software bots in general are useful for automating many different tasks, prior research has shown that they have the potential side-effect of disrupting developers in their work [50, 53]. However, there is a lack of investigation to determine whether that is the case for dependency bots and what types of overhead these specific types of bots introduce. Wessel et al. [52] found that package maintainers complain that bots in open source software (OSS) projects provide incomprehensive or poor feedback on pull requests, and that they are often overwhelmed with notifications, thereby increasing the level of effort required to address any issues created by the bot. Therefore, we consider *overhead* in the context of dependency management as referring to the need for developers to address issues or recommendations created by the bot in their projects. This includes any form of notification that requires developer's attention, and may consist of a significant amount of noise (e.g., if the GKIR is created as a result of some issue unrelated to the dependency update, and therefore is a false alarm).

To that end, we explore the overhead that is introduced in clients who use tools like Greenkeeper. Specifically, we investigate 1) how prevalent are GKIRs in client projects and what are the artifacts (e.g., comments) created as a result of these GKIRs (Section 4.1.1)?, 2) how long does it take for clients to address these artifacts (Section 4.1.2)?, and 3) are these artifacts actually useful to clients (Section 4.1.3)?

4.1.1 How prevalent are GKIRs in client projects and what are the artifacts (e.g., comments) created as a result of these GKIRs?

Approach. We first examine the proportion of studied projects' issues that are GKIRs, beginning from the point in time when Greenkeeper first created a GKIR in each project. We use this point in time as a proxy for when each project first adopted Greenkeeper. Exploring this metric can provide a sense of how prevalent Greenkeeper is in projects that adopt it. We aim to reduce any bias introduced by projects with a low number of IRs, as these cases may skew the proportion of GKIRs in a project (e.g., a project with only 3 IRs, 2 of which are GKIRs will have a proportion of two-thirds). Therefore, for this RQ, we first calculate the median number of IRs for projects in our dataset, and then specifically analyze projects that have at least the median number of total IRs.

If an IR has been closed, it is an indicator that someone (e.g., a human developer or a configured bot) has decided that either the issue has been fixed, or that the issue is not, in fact, a problem for the project in question (i.e., a false alarm), and can be closed. Therefore, we consider any GKIRs that have been closed to be resolved. We examine both the overall proportion of GKIRs that have been closed versus those that remain open, as well as each individual project's proportion of GKIRs that have been closed. These metrics can provide a sense of how much attention is required by Greenkeeper from package maintainers compared to the rest of the project. We compare each project's proportion of closed GKIRs to non-GKIRs to discover whether GKIRs are resolved at a higher rate.

Findings. Observation 1) *Half of the IRs in projects that have integrated with Greenkeeper are GKIRs.* This represents a very large proportion of IRs to be created by a single bot. To account for packages with a significant number of IRs in our results, we perform the same analysis only on projects that have at least the median number of total issues. We find that 41.7% of IRs in these projects are GKIRs, which is still a high percentage of a project's IRs to be created by a bot. The distributions of the proportion of GKIRs per project are shown in Figure 6. This observation suggests that Greenkeeper is very prevalent in client projects that adopt it, and requires much attention from client developers.

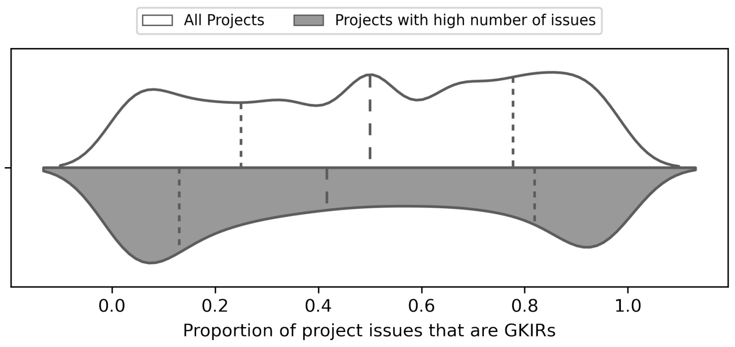


Fig. 6. Violin-plot showing the distribution of the proportion of project issues that are in-range breaking build update issues. The dashed lines indicate the first quartile, median, and third-quartile.

Observation 2) *Clients close approximately the same proportion of GKIRs as other issues in a project.* We consider a GKIR being closed to indicate that it was determined that the GKIR has either been fixed or the GKIR is not a problem. Overall, we observe that 82.3% of GKIRs are closed (i.e., resolved), compared to 79.8% of non-GKIRs. This high proportion of closed GKIRs indicates that developers are highly responsive to these issues, as a developer would have had to determine that the GKIR has either been fixed or the GKIR is not a problem in order to close the GKIR. Of the 17.7% of GKIRs that are not closed, we find that 99% do not have any form of interaction from a client developer (e.g., a comment or a referenced commit), indicating that these GKIRs are simply ignored by the client developers.

4.1.2 How long does it take for clients to address these artifacts?

Approach. There is incentive for clients to resolve GKIRs in a timely manner, especially if the offending dependency is a runtime dependency, as users of the client's project will be affected by the GKIR while the issue remains unaddressed, being unable to successfully build or install the client project. However, this issue is only relevant to clients who have dependent projects. Therefore, for this analysis, we only examine projects that have at least 10 stars on GitHub, which is a measure of package popularity [8], and whose package name is available on the npm registry, which indicates that the client package is also available as a provider package for other projects to depend on. After applying this filter, we find that 32.7% (3152) of the client projects in our dataset meet this criteria. For these projects, we analyze the amount of time it takes for GKIRs to be resolved (closed) compared to non-GKIRs. To do this, we calculate the distribution of the time difference (in days) between the creation date and the close date for closed GKIRs and non-GKIRs for each project.

We compare the two distributions for each project and verify whether they are statistically different. We test the null hypothesis that both distributions do not differ from each other after using the Wilcoxon Rank Sum test ($\alpha = 0.05$) [5] and correct the resulting p values using Bonferroni type adjustment [4]. For statistically significant distributions, we assess the magnitude of the difference with the Cliff's Delta (d) estimator of effect size [12]. To classify the effect size, we use the following thresholds [47]: negligible for $|d| \leq 0.147$, small for $0.147 < |d| \leq 0.33$, medium for $0.33 < |d| \leq 0.474$, and large otherwise. We report the proportion of associated projects with each effect size, as well as the distributions of the median time-to-close GKIRs and non-GKIR per project.

Findings. Observation 3) ***Popular projects that are available as provider packages take a median of 6 days to resolve GKIRs, which is in line with non-GKIRs.*** During these 6 days, users of these popular client projects could potentially be affected by the issue that caused the GKIR, which is a considerably long time for a package to be in a broken state. We consider whether clients may resolve GKIRs faster depending on if the offending dependency was a runtime dependency or a development dependency. Approximately three-quarters of GKIRs were opened for updating a provider package that was a development dependency of the client. This means the dependency is not required by the client project in production, and the client's users will not be affected by any issues caused by the dependency. GKIRs for these types of dependencies are resolved in a median of 6.5 days. The remaining quarter of GKIRs were for runtime dependencies, which are required by the client in production. In these cases, if the GKIR was indeed caused by the dependency, then new installs of the client project will fail because of the new dependency release. GKIRs for these types of dependencies are resolved in a median of 5.71 days. While the difference between the time taken to close GKIRs opened for development or runtime dependencies is statistically significant ($p < 0.05$), the effect size is negligible ($|d| = 0.035$), implying that the type of dependency that caused the GKIR does not affect how fast client developers take to resolve these issues.

When comparing projects' median time taken to close GKIRs and non-GKIRs, we find that the vast majority (98%) of the distributions are not statistically significant or have a negligible effect size. This implies that developers tend to resolve GKIRs at the same speed as non-GKIRs in their projects.

4.1.3 Are these artifacts actually useful to clients?

Approach. To explore the notifications that clients receive in addition to the notification caused by the creation of a GKIR, we examine the activity that occurs on GKIRs in the form of comments and events. We use specific patterns (Table 1) that are used by the bot at the time of the data collection to match types and frequency of comments made by Greenkeeper on GKIRs (see Section 2.3).

Table 1. String patterns for classifying types of Greenkeeper comments.

Comment Type	**String Pattern**
Failing new release	*Your tests are still failing with this version*
Passing new release	*Your tests are passing again with this version*
Failed pin attempt	*^After pinning to .* your tests are still failing*
Passing pin attempt	*^After pinning to .* your tests are passing again*

The comments left by users on GKIRs provide a unique source of information, as developers may provide their rationale for considering the GKIR as resolved before closing the IR. We use the `user_type` attribute on the comment records to distinguish between comments left by users and comments left by bots.

We lemmatize the comments left by users on GKIRs, and initially set each lemmatized comment body in the full data set as unclassified. The following steps are then used to classify the comments: 1) the first author manually examined a sample size of 50 unclassified comment bodies from the full data set to extract common patterns that could be grouped into similar categories, 2) these new patterns are added to a set of regular expressions, 3) the full data set of lemmatized comments are then re-classified with the updated regular expressions, 4) the process is repeated until any new extracted patterns do not classify a threshold of at least 1% of the unclassified comments. Once this threshold was reached, 74.8% of the comments in the full data had been matched to one of following three overarching categories: *Referenced Fix* (i.e., the comment indicates the GKIR has been resolved), *False Alarm* (i.e., the comment indicates the GKIR is a false alarm), and *Tool Mentioned* (i.e., the comment mentions Greenkeeper, the CI system, or some other tool used by the client developers). Each overarching category consists of multiple sub-categories, the patterns for which are shown in Table 2. We then examine the proportion of each of these categories as a lower bound estimate of how often developers are responding to GKIRs due to a dependency problem or are indicating the GKIRs to be false alarms.

Table 2. String patterns for classifying "False Alarm" and "Fix Referenced" user comments on GKIRs.

Category	**Sub Category**	**Regular Expression**
Fix Referenced	PR URL	`https:\/\/github\.com\/[\S]*\/(pull)\/[\S]*`
	Closed By	`((closed|fixed|resolved|done|updated)( in|by|via|with)+ #\d*)`
	PR/Commit Number	`(^#\d*|(merged|close|pr|see).*#\d*)`
	Fix Mentioned	`(resolve|fix|bump|merge|upgrade|done|clos(e|ing)|solved)`
False Alarm	Flaky	`flake|flaky|flakiness|fluke|unrelated`
	Inconsistent	`(inconsistent|brittle|unstable|spurious( unit)?) test`
	Build Hiccup	`(server|test|CI|build) (hiccup|is actually passing|failed for other reasons)`
	Random Failure	`(random|intermittent)( build|test|CI)? (failure|error)`
	Rerun Pipeline	`re-?(run|ran|starting|build|tried|trigger|start)`
	False Positive	`false (positive|alarm|negative|alert|flag)|invalid|non-issue|no action required|obsolete(d)?|not relevant`
	Timeout	`time-?out`
Tool Mentioned	Mention CI System	`(Travis|CircleCI|Cirecl CI|Jarvis|Jenkins|BitHound|CI.*issue)`
	Mention Greenkeeper	`Greenkeeper`
	Mention **Renovate**	`Renovate`

Findings. Observation 4) *GKIRs generate a significant amount of noise in client projects.* The vast majority (96.8%) of comments on GKIRs are from the Greenkeeper bot itself. 80% of GKIRs have an initial comment from Greenkeeper reporting the status of attempting to pin the dependency, and GKIRs in general have a median of 2 comments from Greenkeeper. Figure 7 shows that the longer a GKIR remains open, the higher the likelihood that it will continue to generate notifications in the client project, as Greenkeeper will comment on the existing GKIR while the GKIR remains open whenever the provider releases a new version, rather than creating a new GKIR.

In total, 38% of GKIRs remain open long enough to see a new release from the provider. Of these GKIRs, approximately four out of five (81.3%) only see new releases that continue to fail the client's tests. This means that if the GKIR is a false alarm (i.e., the GKIR was not in fact caused by the dependency being updated, but rather some unrelated issue with the client's project) the client will constantly receive notifications that their build continues to fail with new dependency updates until they determine the GKIR is, in fact, a false alarm and that they can safely close the issue.

Observation 5) *Developers tend to not comment on GKIRs, but when they do, they usually indicate the issue has been resolved or is a false alarm.* Only 9.3% of GKIRs have a comment from a developer, versus 74.6% of non-GKIRs from the same set of projects. We classify these comments (Figure 8) using the method described in the approach, and report the proportion that indicates the GKIR has been fixed and how many indicate the GKIR is a false alarm, which are the two most common overarching categories. We found that approximately half (47.8%) of these

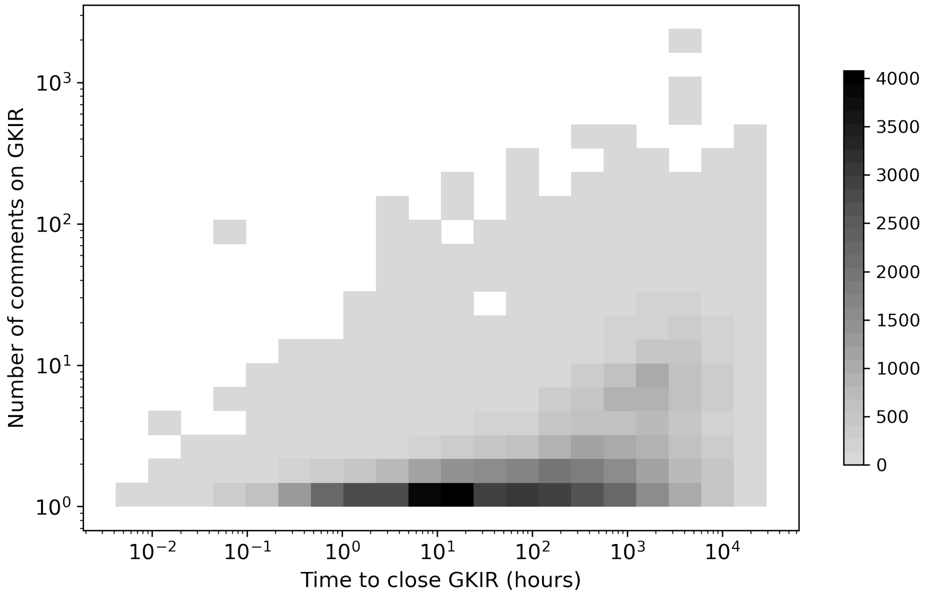


Fig. 7. The distribution of the time taken to close a GKIR against the number of comments on the GKIR. The bin colour indicates the frequency of each data point.

developer comments are referencing a fix for the GKIR. For example, users may reference a PR (e.g., *"Fixed with PR #169."*), a commit (e.g., *"Fixed via f6800c7"*), or simply say that the issue has been resolved(e.g., *"Fixed manually"*).

Additionally, we found that one in five (19.8%) developer comments indicate that the GKIR is a false alarm. For example, users indicate that the GKIR was caused by the CI pipeline (e.g., *"The tests passed after re-running the Travis build."*, *"This is a false positive, the build had timed out."*), that their project's tests failed for a non-deterministic reason (e.g., *"Flaky test"*), or simply that the GKIR is not, in fact, an issue (e.g., *"False positive"*).

RQ1: What is the overhead introduced in client projects by Greenkeeper?

- Greenkeeper induces a significant amount of overhead that must be addressed by clients, with GKIRs making up half (50%) of the IRs in all projects, or two-fifths (42%) in projects with a high number of IRs.
- The vast majority (97%) of comments on GKIRs are from Greenkeeper, with more comments continuing to be generated by Greenkeeper the longer the GKIR remains open, creating further notifications in client projects.
- Nearly one-fifth (19%) of user comments on GKIRs indicate that the GKIR is a false alarm, meaning the GKIR only serves to create noise in the client project.

4.2 RQ2: Is automated dependency pinning an effective mechanism for resolving GKIRs?

Motivation. Greenkeeper's automatic pin attempt feature is an interesting phenomenon that deserves to be investigated further, as automatically attempting to pin the dependency as a best-effort solution has the potential to make the client package "downloadable" again quite quickly, with minimal effort on the part of the client developers. It stands to reason that the pinning attempt should succeed the majority of the time, since if a client's build was passing before a

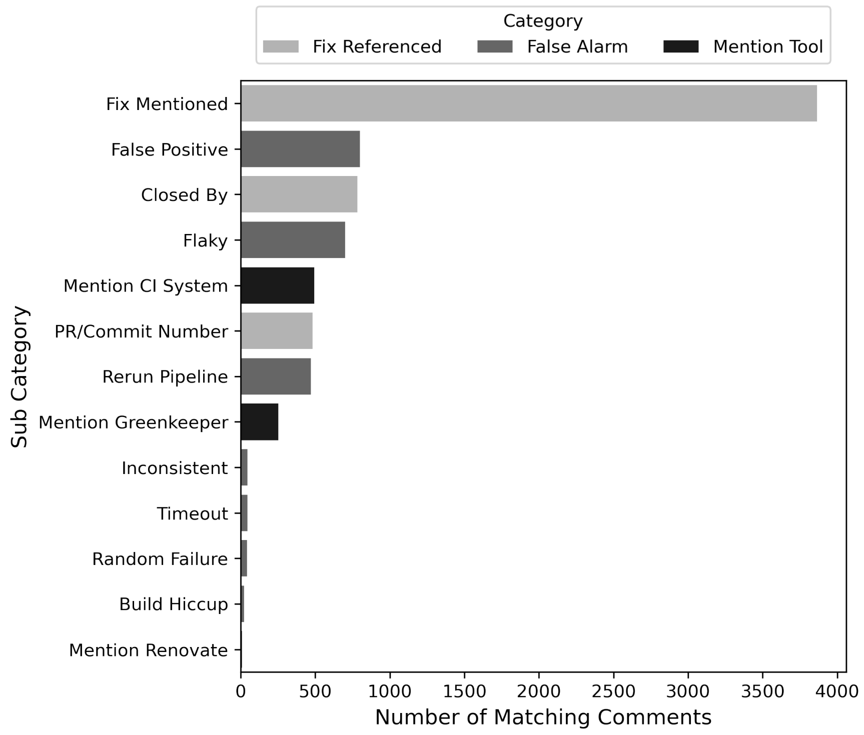


Fig. 8. Bar-plot showing the number of comments matched by each patterns shown in Table 2.

dependency released a new version that broke the client's build, pinning the dependency back to the prior version should result in the client's build passing again. Yet, pinning should only be a temporary measure, as the client will no longer receive bug fixes or security updates from the provider.

However, if the pin attempt fails, then the client developer's attention is required to address the GKIR. In this case, the overhead of resolving the associated problem with the GKIR will incur on the client developers. Additionally, a failed pin attempt may be a good indicator that the GKIR was not in fact caused by the dependency being updated, but rather by some other issue that was already present in the client's project. Therefore, in this RQ, we investigate the efficacy of Greenkeeper's automatic pinning feature and the types of issues developers need to address when the pinning attempt fails.

Approach. We look at the effectiveness of Greenkeeper's automatic attempts at pinning the offending dependency to resume passing the client's CI pipeline. Greenkeeper posts a comment following one of two specific patterns to notify the user whether the pin attempt was successful. We determine whether the automatic pin attempt was successful by searching for comments on GKIRs from Greenkeeper that match the pinning status patterns shown in Table 1.

To explore the types of issues that need to be addressed by client developers when Greenkeeper's automatic pinning attempts fail, we manually analyze a statistically significant sample (95% confidence level and ±5% confidence interval) of GKIRs that have a failed pin attempt by Greenkeeper (381 cases out of 51,720). For each GKIR with a failed pin attempt, we check whether the build logs for the CI pipeline that failed are available. Whenever an observation in our sample did not meet this requirement, we randomly drew another observation from the population of GKIRs with a failed pin attempt. We then categorize the build logs to determine the reasons the client CI pipelines failed for the dependency updates that was followed by a failed pin attempt.

To mitigate the risk of the classifications being biased by the first author, both the first and second authors independently classified 15% (58) of the random samples, first examining the build logs of each sample and (if possible) summarizing the reason for why the build failed in one sentence. Each author then independently extracted common categories for why the builds failed, and then discussed their individual categories and consolidated the classes. During the discussion, there were 15 cases for which the only difference was the level of granularity with which we described the class (e.g., in a case where the client's build failed because the test suite timed out, one author considered the class to be a test failure, while the other considered the class to be a timeout error) — we consolidated these cases into classes that were commonly agreed. There were three more cases for which the two authors disagreed (e.g., one authored mistakenly attributed a build failure to a syntax error, when in fact a missing dependency caused the failure). The three disagreements were discussed and an agreement was reached for them. Through this manual analysis and following discussion, 10 different categories for created GKIRs with a failed pin attempt were identified.

Considering the existence of 10 categories and three disagreements out of 58 analyzed cases, we calculate the inter-rater agreement in our methodology using Cohen's Kappa coefficient [14]. The Cohen's Kappa coefficient has been used to evaluate inter-rater agreement levels for categorical scales, and provides the proportion of agreement corrected for chance. The resulting coefficient is scaled to range between -1 and +1, where a negative value means less than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement [24]. In our case, the level of agreement is +0.93, which indicates that the classification results made by the first author are more likely to hold [31, 49]. Taking this high level of agreement into account, the first author then classified the remaining 85% of the random sample using the 10 categories agreed upon by both the first and second authors, which is a common process and has been done in previous work [20, 21, 36].

We found in the previous RQ that only approximately 1 in 5 GKIRs that see a new dependency release actually have their client's tests resume passing with the new release. In this RQ, we extend this analysis to explore how often a new release from the dependency is able to resume passing the client's tests on a GKIR that has a failed pin attempt, as a failed pin attempt on a GKIR could indicate that Greenkeeper should limit its attempts to test new dependency releases on opened GKIRs. Greenkeeper will comment on GKIRs whether a new release of the dependency is able to resume passing the client's tests. We determine these comments by matching the comment body against the patterns shown in Table 1 for alerting the client of a new dependency release.

Findings. Observation 6) ***The vast majority of the unsuccessful pin attempts are unrelated to the dependency and require a manual intervention from the client developers.*** 80.3% of GKIRs have a pinning attempt. 3.3% of GKIRs are for bundled dependency updates, which Greenkeeper does not perform any pin attempt on, and it was not clear why no pin attempt was performed by Greenkeeper on the remaining 16.4%.

Regarding the overall proportion of automatic pin attempts on GKIRs, we observe that only 32% are able to successfully resume passing the client's tests. This finding was surprising, since, in principle, the updated dependency is the only difference between the GKIR branch and the project's main branch, and pinning the dependency to the previous version (i.e., the version that was previously in use on the project's main branch) should in effect render the GKIR branch a duplicate of the project's main branch. Therefore, we expected the majority of pinning attempts to be successful.

However, this was not the case, as over two-thirds of pinning attempts fail, which implies the client's build was already broken for an unrelated reason to begin with, and that new installs of the client project may be failing as well.

To further investigate why so many pin attempts were failing, we manually analyze a statistically significant sample of GKIRs that have a failed pin attempt using the process described in the approach. The categories are explained below, and are summarized in Table 3.

Table 3. Prevalence and description of reasons for created GKIRs with a failed pin attempt.

ID	Category	Proportion	Description
C1	Syntax/Linter/Project Guideline Error	17.6%	The client's CI failed because of a syntax or linter error in the source code, or some other requirement for the project was not met (e.g., code coverage).
C2	Client Test Case Failure	13.6%	The client's CI failed because of an assertion error in the client's test suite.
C3	Incompatible Node/NPM/Dependency Version Error	13.4%	The client's CI failed because an invalid version of either Node, npm, or one of their dependencies was specified.
C4	Dependency Error	13.1%	The client's CI failed because one of their dependencies threw an error (not related to the dependency being updated).
C5	Missing File/Module	11.3%	Either a file or an entire module was missing from the client's CI environment, causing it to fail.
C6	Lockfile Error	10.8%	The client's CI failed because the client's *package.json* and the associated lockfile were out-of-sync.
C7	Client Tests Failing to Run Successfully	10.0%	The client's CI failed because the test suite encountered an internal issue and did not run to completion.
C8	Timeout/Network Error	5.2%	The client's CI failed because either their build process stalled for too long or communication over a network was not successful.
C9	Security Error	2.6%	The **npm audit** command detected a security vulnerability in one of the client's dependencies, causing the CI pipeline to fail.
C10	Invalid Credentials Error	2.4%	The client's CI failed because the build environment had invalid credentials, or was missing them entirely.

• **C1: Syntax/Linter/Project Guideline Error (17.6%):** Client's source code may have existing syntax errors that cause the CI pipeline to fail, or a linter can fail a build if any of the code in the project does not meet the style guidelines set by the client. Listing 1 shows an example of a client's build[23] failing because of multiple style rule infractions. Additionally, a build can be configured to fail if the bundle size of the project grows too large or the test coverage drops below a specific threshold.

[23]https://travis-ci.org/github/cnap-cobre/synapse-frontend/jobs/490193979

Listing 1: Example error snippet of build logs with a linter error.

```
...
npm run lint
> synapse-frontend@0.3.4 lint /home/travis/build/cnap-cobre/synapse-frontend
> eslint src
/home/travis/build/cnap-cobre/synapse-frontend/src/components/FavoritesBar/FavoritesBar.js
73:6   error  Missing semicolon  semi
/home/travis/build/cnap-cobre/synapse-frontend/src/store/files/sagas.js
  4:35  error  Missing semicolon  semi
130:33  error  Missing semicolon  semi
164:52  error  Missing semicolon  semi
4 problems (4 errors, 0 warnings)
4 errors and 0 warnings potentially fixable with the `--fix` option.
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! synapse-frontend@0.3.4 lint: `eslint src`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the synapse-frontend@0.3.4 lint script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.
npm ERR! A complete log of this run can be found in:
npm ERR!     /home/travis/.npm/_logs/2019-02-07T18_47_18_800Z-debug.log
The command "npm run lint" exited with 1.
...
```

• **C2: Client Test Case Failure (13.6%):** A client's tests can fail for reasons unrelated to the dependency update, either due to some existing issue or perhaps a flaky test. For example, log output may not match what is specified to be expected in the test, or some other assertion test may evaluate to false, causing the client's tests to fail, as is shown in the excerpt of the build logs[24] of Listing 2.

Listing 2: Example error snippet of build logs where a failure of the client's test case caused the CI pipeline to fail.

```
...
npm test
> @slimio/timemap@0.3.0 test /home/travis/build/SlimIO/TimeMap
> cross-env psp && ava --verbose
> Running Project Struct Policy at /home/travis/build/SlimIO/TimeMap
Finished with: 0 Criticals and 0 Warnings
1 test failed
construct new TimeMap
test/test.js:23
22:     assert.is(Object.keys(map).length, 3);
23:     assert.is(Reflect.ownKeys(map).length, 6);
24:
Difference:
- 7
+ 6
npm ERR! Test failed.  See above for more details.
The command "npm test" exited with 1.
...
```

[24]https://travis-ci.com/github/SlimIO/TimeMap/builds/149528914

- **C3: Incompatible Node/NPM/Dependency Version Error (13.4%):** Projects may run their builds with multiple jobs, for example, 1 job each for versions 4, 6, and 7 of Node, but dependencies might require Node >= 6.0.0, which causes one of the build jobs to fail, as is the case in the build logs[25] shown in Listing 3. Additionally, dependency versions can conflict, or sometimes can not be found altogether, again causing the client's CI pipeline to fail.

Listing 3: Example error snippet of build logs where an incompatible version of **Node** was used.

```
...
yarn install v1.3.2
[1/5] Validating package.json...
[2/5] Resolving packages...
[3/5] Fetching packages...
error har-validator@5.1.3: The engine "node" is incompatible with this module. Expected version ">=6".
error Found incompatible module
...
```

- **C4: Dependency Error (13.1%):** One of the client's dependencies used during the CI pipeline can fail because they have not been configured properly. For example, Listing 4 shows the build logs [26] where the client's CI pipeline failed due to a dependency not being initialized properly.

Listing 4: Example error snippet of build logs where an error from a dependency of the client caused the client's CI to fail.

```
...
-----
selenium-standalone installation finished
-----
wdio
/home/travis/build/chmanie/wdio-intercept-service/node_modules/@wdio/utils/build/initialisePlugin.js:19
    throw new Error(Could not initialise "\${name}".\n\${e.stack});
    ^
Error: Could not initialise "@wdio/local-runner".
Error: You can not reassign a plugin after applying another plugin
...
error Command failed with exit code 1.
Done. Your build exited with 1.
...
```

- **C5: Missing File/Module (11.3%):** A client's CI pipeline can fail because of missing files or even entire modules. We found cases where the initial clone of the project failed, resulting in failed builds, as well as situations where dependencies were not available. Listing 5 shows an example excerpt of a client's build logs[27] where the build configuration file is missing completely, automatically causing the CI pipeline to fail.

[25] https://travis-ci.org/github/jaumard/trailpack-acl/jobs/529853292
[26] https://travis-ci.org/github/chmanie/wdio-intercept-service/jobs/526469907
[27] https://app.circleci.com/pipelines/github/unional/clibuilder/1/workflows/23256a75-d014-4d80-acf0-842644bfae24/jobs/1719

Listing 5: Example error snippet of build logs where the project was missing a configuration file.

```
...
#!/bin/sh -eo pipefail
# No configuration was found in your project. Please refer to https://circleci.com/docs/2.0/ to get started
     with your configuration.
#
# -------
# Warning: This configuration was auto-generated to show you the message above.
# Don't rerun this job. Rerunning will have no effect.
false

Exited with code exit status 1
CircleCI received exit code 1
...
```

- **C6: Lockfile Error (10.8%):** Greenkeeper will bump the version specification in the *package.json* file for the dependency being updated, then run the client's CI pipeline. However, clients can specify in their CI install script the `–frozen-lockfile` flag, which results in the *package.json* file and the associated lockfile becoming out of sync, as early versions of Greenkeeper were not able to update the lockfile[28], causing the build to fail. Greenkeeper has since added native support for this feature[29]. However, this remains a common issue, as shown in the build logs[30] of Listing 6 as an example.

Listing 6: Example error snippet of build logs with a out-of-sync *package.json* and *package-lock.json* error.

```
...
npm ci
npm ERR! cipm can only install packages when your package.json and package-lock.json or npm-shrinkwrap.json are
      in sync. Please update your lock file with `npm install` before continuing.
...
```

- **C7: Client Tests Failing to Run Successfully (10.0%):** While this category is similar to **C2: Client Test Case Failure**, we differentiate the two because, in **C2**, the client's tests fail due to some assertion error, whereas in this category the client's test suite does not run to completion due to an issue. For example, Listing 7 shows the build logs[31] of a client's test suite failing to run because of an error it the client's testing code.

[28]https://blog.greenkeeper.io/greenkeeper-and-lockfiles-a-match-made-in-heaven-8260943fe521
[29]https://blog.greenkeeper.io/announcing-native-lockfile-support-85381a37a0d0
[30]https://travis-ci.org/github/travi/hapi-react-router/builds/619316527
[31]https://travis-ci.org/github/G5/gtm-controller/builds/463979922

Listing 7: Example error snippet of build logs that show a client's test suite not being able to run to completion.

```
...
npm run test:coverage
> @g5/gtm-controller@1.0.0 test:coverage /home/travis/build/G5/gtm-controller
> cross-env JEST_COVERAGE=true jest
PASS __tests__/triggers/iframeFocusTrigger.test.ts
PASS __tests__/helpers/ruleParser.test.ts
PASS __tests__/core/dataLayer.test.ts
FAIL __tests__/triggers/trigger.test.ts
- Test suite failed to run
  TypeScript diagnostics (customize using `[jest-config].globals.ts-jest.diagnostics` option):
  __tests__/triggers/trigger.test.ts:3:30 - error TS2314: Generic type 'Trigger<SubscriptionDataType>' requires
      1 type argument(s).
...
```

• **C8: Timeout/Network Error (5.2%):** The project's build may not receive any output for a specified threshold of time, in which case the build will time out and be marked as failed. Listing 8 shows an example excerpt of a client's build logs[32] with this scenario. Additionally, network requests (e.g., download a dependency or upload code coverage statistics) can fail.

Listing 8: Example error snippet of build logs with a build timeout error.

```
...
Downloading https://nodejs.org/dist/v8.16.0/node-v8.16.0.tar.xz...
...
No output has been received in the last 10m0s, this potentially indicates a stalled build or something wrong
      with the build itself.
Check the details on how to adjust your build configuration on: https://docs.travis-ci.com/user/common-build-
      problems/#Build-times-out-because-no-output-was-received
The build has been terminated
...
```

• **C9: Security Error (2.6%):** The `npm audit` command will return a failure code if any security vulnerabilities are detected in any of the project's dependencies. For example, if a vulnerability is found in a dependency other than the offending dependency that caused the GKIR (or the offending dependency for that matter), the `npm audit` command will still cause the pipeline to fail. Listing 9 shows an example excerpt of a client's build logs[33] with this scenario.

Listing 9: Example error snippet of build logs with a security error.

```
...
npm audit
...
found 1 high severity vulnerability in 12912 scanned packages
run `npm audit fix` to fix 1 of them.
The command "npm audit" failed and exited with 1 during .
Your build has been stopped.
...
```

[32]https://travis-ci.org/github/visusnet/typereact/builds/549691105
[33]https://travis-ci.com/github/r3nya/r3nya.github.io/builds/104464336

- **C10: Invalid Credentials Error (2.4%):** Clients often need to specify credentials in their CI environment to allow authenticated actions (e.g., cloning the project, pushing test results to an external repository, etc.). However, these credentials may become invalid or be missing entirely from the CI environment, which can cause the CI pipeline to fail. Listing 10 shows an example excerpt of a client's build logs[34] where the CI environment does not have the correct access rights to clone the project.

Listing 10: Example error snippet of build logs that error because of missing or invalid credentials.

```
...
Using SSH Config Dir /home/circleci/.ssh
Cloning into '.'...
Warning: Permanently added the RSA host key for IP address '140.82.113.3' to the list of known hosts.
Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

exit status 128
CircleCI received exit code 128
...
```

Observation 7) ***Fewer than 1 in 10 GKIRs that have a failed pin attempt and eventually see a new dependency release have their build resume passing with the new dependency release applied.*** The results from Greenkeeper's automatic pin attempt appear to be a good indicator of whether the GKIR is a false alarm. 91% of GKIRs that have a failing pin attempt and stay open long enough to see at least one new release from the dependency never have their tests resume passing again due to Greenkeeper attempting to update the dependency on the GKIR. In other words, if the initial pin attempt on a GKIR is not successful, any subsequent attempts by Greenkeeper to attempt to fix the GKIR by upgrading to a new release of the dependency will also most likely fail, and only serve to flood the client's project with redundant notifications.

RQ2: Is automated dependency pinning an effective mechanism for resolving GKIRs?

- Greenkeeper's automatic pinning attempts have a failure rate of 78%, which is surprising, as pinning the dependency should render the GKIR branch a duplicate of the projects main branch.
- GKIRs with a failed pin attempt are usually caused by an error with the client's CI pipeline (e.g., syntax error, incompatible Node version, etc.), rather than the new dependency release, which means these GKIRs can be considered false alarms from the perspective of being a dependency issue.
- 91% of GKIRs that have a failed pin attempt and are open long enough to see a new release from the dependency never have their tests resume passing again due to Greenkeeper attempting to update the dependency on the GKIR, and only serve to flood the client's project with redundant notifications.

4.3 RQ3: What are the performed code changes when resolving GKIRs?

Motivation. While pinning the breaking dependency to its previous working version may be the quickest and easiest method to resolve the issue, it is not always successful, and in fact is considered an anti-pattern for dependency management [28], as this type of versioning specification is often associated with outdated dependencies [56]. However,

[34]https://app.circleci.com/pipelines/github/gucong3000/gulp-reporter/7/workflows/131b9bbd-51b4-4131-be90-cf92603a3790/jobs/1045

pinning is not the sole method available to resolve GKIRs, and clients may prefer other more complex strategies that allow them to continue taking advantage of the benefits of using a version range for the dependency.

Ideally, minimal changes would be needed to resolve a GKIR while maintaining the project's updatability and resolving the issue in a timely manner. For this reason, it is important to explore the code changes (other than pinning) that are performed when resolving GKIRs, which is what we examine in this RQ.

Approach. First, we examine the proportion of file types that are most often modified, as well as the size of the modifications that clients are pushing to resolve GKIRs. To do this, we collect the patch diff from any commits that are referenced by GKIRs. Specifically, we look at the number of files changed in the commit, as well as the lines of code (LOC) churn in the commit (i.e., added lines + removed lines), which are metrics that have been used in previous work to measure the impact of code changes [40, 43]. For example, if a commit only modifies a single LOC (e.g., rename a variable), the churn metric would have a value of 2 (i.e., 1 addition and 1 deletion). To compare these changes against a baseline, we perform the same analysis on commits referenced by non-GKIRs from the same projects. To select the commits for our baseline, we find the preceding non-GKIR that references a commit for each GKIR that references a commit, and compare the metrics for these two distributions of commits.

We anticipate that the majority of commits referenced from GKIRs will contain modifications to the project's *package.json*, as Greenkeeper is a dependency management bot and this file contains the client's dependency specifications. Therefore, we additionally parse the changes made specifically to the *package.json* file to explore how clients are modifying their dependency version specifications in order to resolve GKIRs. We extract the modifications made to the client's dependency specification files and parse the previous and current dependency specification version using the semver[35] package. We then compare the previous and current dependency specifications to determine whether the dependency was updated, downgraded, pinned, added, or deleted. We use this information to learn the most common strategies used by clients for resolving GKIRs that only modify their dependency specifications, which would be simple solutions that dependency management bots like Greenkeeper could automatically implement, potentially reducing the overhead on client developers.

Additionally, for commits referenced by GKIRs that modify more than just the client's dependency specification files, we examine the most common file types that are changed when resolving GKIRs. We again perform the same analysis on commits referenced by preceding non-GKIRs from the same projects.

Findings. Observation 8) ***The changes required to resolve GKIRs are similar to that of non-GKIRs.*** When comparing the number of file changes in commits referenced by GKIRs and non-GKIRs, we find that they both change a median of 2 files per commit. Figure 9 shows the distribution of the number of files changed in commits referenced by GKIRs and commits referenced by non-GKIRs immediately preceding GKIRs. The difference between the two distributions is statistically significant ($p < 0.05$), however the effect size is negligible ($|d| = 0.23$).

Similarly, the size of the changes in the commits referenced by GKIRs and non-GKIRs is comparable, with commits referenced by GKIRs having a median of 33 LOC churn and commits referenced by non-GKIRs having a median of 38 LOC churn. Figure 10 shows the distribution of the number of LOC churn in commits referenced by GKIRs and commits referenced by non-GKIRs immediately preceding GKIRs. The difference between the two distributions is not statistically significant ($p > 0.05$).

Observation 9) ***More than half (56%) of changes that resolve GKIRs only include changes to dependency specification files.*** These changes are primarily made to the *package.json* file, which is manually maintained and is the

[35]https://pypi.org/project/semver/

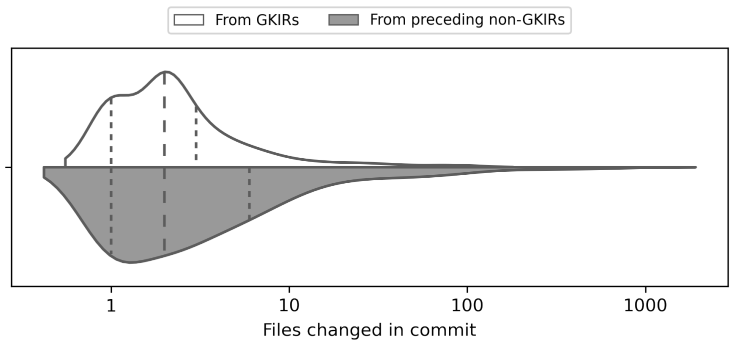


Fig. 9. Violin-plot showing the distribution of number of files changed in commits referenced from GKIRs and commits referenced from non-GKIRs immediately preceding GKIRs. The dashed lines indicate the first quartile, median, and third-quartile.

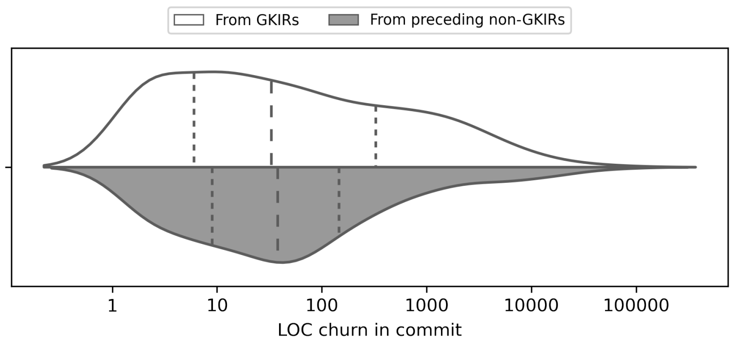


Fig. 10. Violin-plot showing the distribution of lines of code (LOC) churn in commits referenced from GKIRs and commits referenced from non-GKIRs immediately preceding GKIRs. The dashed lines indicate the first quartile, median, and third-quartile.

most common file to be changed, being modified in 78% of commits referenced on GKIRs, versus only 17% of commits referenced from non-GKIRs. Additionally, the npm *package-lock.json* file and its similar counterpart *yarn.lock*, appear in 28% and 27% of all referenced commits, respectively, versus just 4% and 3% of commits referenced by non-GKIRs, respectively. However, these files are automatically generated whenever a project's dependency specifications change, and therefore changes to these files do not indicate any significant overhead introduced on the client developers.

Changes that resolve GKIRs by only modifying dependency specification files tend to be small, similar to changes from non-GKIRs. 57% of these commits on GKIRs that modify the *package.json* file are only one-line changes, while 75% modify 3 or fewer lines. Similarly, commits on non-GKIRs that modify the *package.json* file are approximately the same size, with 66% being one-line changes and 81% modifying 3 or fewer lines.

Figure 11 shows the proportion of dependency change types made by clients when modifying the *package.json* file to resolve a GKIR. We found that 88.8% of the dependency specification changes are dependency version upgrades (e.g., bumping a dependency specification from `^1.2.3` to `^1.2.4`), 7.8% are removing the range specification of the dependency, effectively adopting the pinning action suggested by Greenkeeper (e.g., changing the dependency specification from `~1.2.0` to `1.2.0`), 1.6% are adding new dependencies, 1.2% are deleting dependencies, and less than 1% are downgrades (e.g., changing a dependency specification from `1.2.3` to `1.2.1`).

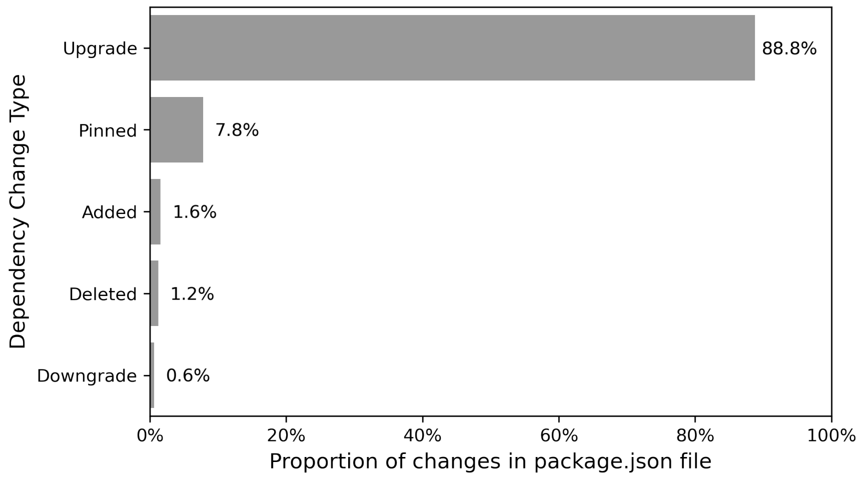
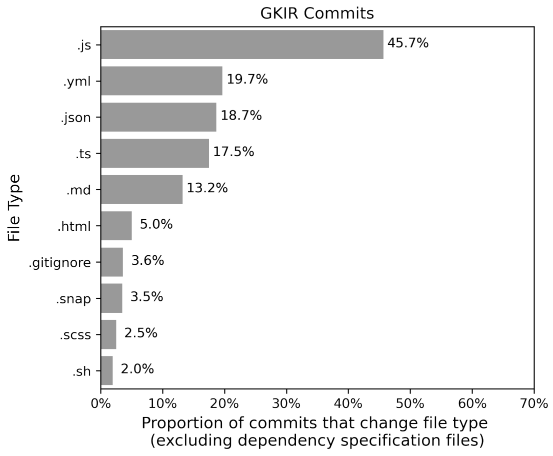


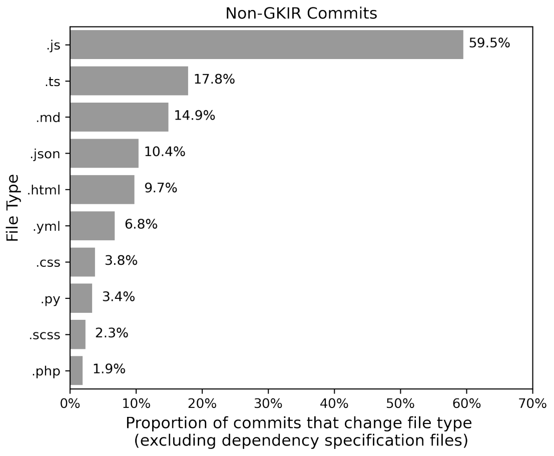
Fig. 11. Bar-plot showing the proportion of the dependency change types made to the *package.json* file on commits referenced from GKIRs.

Additionally, we observe that clients may resolve multiple GKIRs in their project with a single patch. While the majority (79%) of commits only reference a single GKIR, 21% of commits resolve at least 2 GKIRs. We also found that a quarter of commits upgrade at least four dependency version specifications, further suggesting that clients might wait to perform all of their project dependency updates in a batch fix.

Observation 10) ***Commits referenced by GKIRs that do not only modify dependency specification files tend to include changes to a mixture of different file types, similar to commits referenced by non-GKIRs.*** While these commits commonly include changes to source code files (e.g., JavaScript and TypeScript files), they can also include changes to project configuration files (e.g., *.eslintrc.json*) and build pipeline files (e.g., *.travis.yml*). In fact, these commits even sometimes include changes to markdown files (e.g., *README.md*) and even the project's *.gitignore* file. Figure 12 shows the 10 most common file types that are changed in commits referenced by GKIRs (12a) and non-GKIRs (12b) that do not only modify the client's dependency specification files (i.e., *package.json*, *package-lock.json*, and *yarn.lock*).



(a) Commits referenced from GKIRs.



(b) Commits referenced from non-GKIRs.

Fig. 12. Top 10 files types that are changed in commits referenced from GKIRs and non-GKIRs that do not only modify the client's dependency specification files (i.e., *package.json*, *package-lock.json*, and *yarn.lock*).

RQ3: What are the performed code changes when resolving GKIRs?

- Commits referenced by GKIRs that require changes to the client's code are comparable in size (median of 2 files changed, 33 LOC changed) to commits referenced to non-GKIRs (median of 2 files changed, 38 LOC changed).
- More than half (56%) of commits referenced by GKIRs only modify dependency specification files.
- 55% of manual changes to dependency specification files only modify a single statement. 88% of manual changes to dependency specification files are upgrading a dependency version specification, while 7% are pinning a dependency.

5 DISCUSSION

In this section, we discuss the findings presented in Section 4. We present a set of practical implications for designers of dependency management bots with the aim of reducing the overhead generated in client projects by these software bots (Section 5.1), as well as the current state-of-the-art in automated dependency management and avenues for future research (Section 5.2).

5.1 Implications

Implication 1) ***Dependency management bots should provide features that allow clients to reduce the amount of activity generated by the bots.*** We found in RQ1 that half of the IRs in client projects are opened by Greenkeeper. This is a high ratio of IRs to be opened by a bot, and can overwhelm client developers with excessive notifications in their projects. We also found that the longer these GKIRs stay open, the more activity is generated on them by the bot, often in the form of comments notifying the client whether a new release of the dependency has resumed passing the client's tests. This feature can generate a high amount of notifications in the client project, especially if the provider package releases new updates often, and does little to help the client with resolving the issue.

Additionally, if the GKIR turns out to be a false alarm, these notifications only serve to distract the client, and may erode their trust in the dependency bot itself if they find they are being bombarded with notifications for issues that turn out to be false alarms. In fact, the number of notifications generated by dependency management bots is already a common complaint amongst developers on forums and IRs. [36, 37, 38, 39] Therefore, we argue that dependency management bots should provide features that allow clients to configure the bot to reduce the amount of activity generated in their projects, and be mindful of the trade-offs associated with each feature in the context of overhead introduced for the client.

For example, one feature that dependency management bots should support is to allow for a project's dependency updates to be bundled into a single PR. The results in RQ1 show that half of the IRs in client projects are opened by Greenkeeper. We also found in RQ3 that clients may manually group updates for multiple dependencies into a single commit in order to resolve a batch of GKIRs, which suggests that clients could benefit from having the dependency updates in these IRs and PRs grouped, so as to reduce the amount of noise created by the bot in client projects. In fact, this issue has been a subject of discussion in at least four IRs [40, 41, 42, 43] in the Dependabot project, another popular automated dependency bot. However, if one of the bundled dependency updates causes the client's CI to fail, it could require a significant amount of manual work on the client's part to determine which dependency caused the problem.

To explore the efficacy of this recommendation, we compare the time required to close PRs opened by Greenkeeper for grouped dependency updates from monorepos and single dependency updates from non-monorepos, and find that monorepo PRs are closed in a median of 1 day 13 hours, while non-monorepo PRs are closed in a median of 1 day 17 hours. The difference between distributions between these two scenarios is not significant ($p > 0.05$), which implies that the speed of monorepo updates is approximately the same as non-monorepo updates, even though multiple dependencies are being updated by them. This suggests that bundling updates could reduce the amount of activity generated by the bot.

To further reduce the amount of manual intervention required by clients to act on PRs opened by dependency management bots, these bots should consider offering an option to auto-merge any dependency updates if the updates meet a set of requirements set by the client. For example, clients may trust certain provider packages they use in their project, and may prefer to have any dependency updates from these packages that pass their CI pipeline to be automatically merged. This functionality would serve to reduce the overhead required by clients to act on PRs that they would have merged anyway, and is a feature that has been requested for multiple bots. [44, 45] However, if a backwards-incompatible update is released by a dependency that is not caught by the client's CI tests, any issues related to the dependency may not be discovered until after the update has been integrated into the project, at which point the effort required to address the issue could be significant.

Additionally, in order to avoid client projects becoming saturated with PRs opened by dependency management bots, these bots should consider providing the option to limit the amount of active PRs they create in client projects. This functionality will help client developers to avoid being overwhelmed by PRs from the bot, and has in fact been discussed

[36]https://github.com/dependabot/dependabot-core/issues/2265
[37]https://github.com/dependabot/dependabot-core/issues/376
[38]https://github.com/dependabot/dependabot-core/issues/2526
[39]https://github.com/dependabot/dependabot-core/issues/1190
[40]https://github.com/dependabot/dependabot-core/issues/2265
[41]https://github.com/dependabot/dependabot-core/issues/376
[42]https://github.com/dependabot/dependabot-core/issues/2526
[43]https://github.com/dependabot/dependabot-core/issues/1190
[44]https://github.com/dependabot/feedback/issues/954
[45]https://blog.mergify.io/replacing-dependabot-preview-auto-merge-feature/

in at least two IRs. [46], [47] However, this may lead to an increase in technical lag of dependencies, as the number of PRs for dependency updates that can be opened at one time will be limited.

Implication 2) ***Dependency management bots should take into account the state of the client's test suite on their main branch when attempting to update dependencies.*** We found in RQ2 that a high number of false alarms are caused by issues that would have already existed on the project's main branch before the dependency update was attempted. For example, after manually analyzing why the client's CI pipeline failed for GKIRs that had a failed pin attempt, we found that nearly one-fifth of the failures were caused by a syntax, linter or project guideline error that would have already been failing the client's main branch, and was not in fact related to the dependency being updated. Therefore, dependency management bots like Greenkeeper should consider the state of the client's test suite on their main branch when opening IRs for new dependency updates. In other words, if the client's main branch is failing and the dependency update fails for the same reason, the dependency update is most likely not the issue, and the bots should delay their analysis until the main branch on the client project is passing again.

An example of an automated dependency management bot that does this well is Dependabot, with its compatibility score feature[48]. Dependabot functions similarly to Greenkeeper, with a compatibility score for each dependency update being calculated as the percentage of client CI runs that passed when updating between relevant versions. However, Dependabot will only include the results from client CI runs that have a previously passing test suite on their main branch. Using this approach, they avoid negatively biasing the scores with failed CI pipeline runs that are not caused by the provider package being updated. Including this type of functionality by default with automated dependency bots would help to reduce the overhead generated by having to filter out false alarm IRs created by these bots as a result of the client's CI pipeline failing for an existing reason.

Implication 3) ***Dependency management bots should provide more detailed information on a pin attempt than simply reporting whether it succeeded or failed.*** Pinning the dependency is a simple solution that bot designers and developers expect to be effective. Since the updated dependency is the only difference between the GKIR branch and the project's main branch, pinning the dependency to the previous version (i.e. the version that was previously in use on the project's main branch) should in effect render the GKIR branch a duplicate of the project's main branch. Thus it is expected that pinning the dependency should resume passing the client's CI pipeline.

However, we saw in RQ2 that this is often not the case. Therefore, dependency management bots should provide more information to the client explaining why the pin attempt failed, rather than simply commenting that the pin attempt was not successful and that the issue might not be related to the dependency. At a minimum, dependency management bots should analyze the CI logs of the client pipeline to determine an overarching reason why the pin attempt failed. The categories we reported in RQ2 for why the client's CI pipeline failed in GKIRs with a failed pin attempt provide a good basis to categorize these failures. Dependency bots could parse the client's build logs and match the output to regular expressions of common error messages, including information on any matched categories in the issue report.

One of the drawbacks of pinning a dependency is that the client will no longer automatically use the most up-to-date version of the provider package, and will begin to increasingly lag behind as the provider releases new versions. To quantify packages lagging behind with respect to using the latest version of their dependencies, Gonzalez-Barahona et al. [27] first proposed the concept of "technical lag", and multiple studies have examined how prevalent technical lag

[46]https://github.com/dependabot/dependabot-core/issues/2158
[47]https://github.com/dependabot/dependabot-core/issues/2189
[48]https://dependabot.com/compatibility-score/

is in different ecosystems [17, 18, 56]. If the pin attempt succeeds, Greenkeeper does a nice job of reducing the amount of technical lag that is potentially introduced when clients decide to take the pinning route. Greenkeeper will pin to the previous version of the dependency, which is better than simply removing the range statement.

For example, if a client specifies they would like to accept the version range of `~1.2.0` from a provider (i.e., only accept *patch* updates), and version `1.2.4` of the provider causes a GKIR to be created in the client's project, Greenkeeper will attempt to pin the dependency to version `1.2.3` rather than version `1.2.0`. Doing so reduced the amount of technical lag introduced by the pinning action from a lag of 4 patch versions to a lag of 1 patch version. Because the client was using a range operator in their dependency constraints, the client would have been implicitly using version `1.2.3` of the provider in their project before the GKIR was created, and therefore pinning to that version should not introduce any new issues.

Implication 4) ***Dependency management bots should provide a more effective incentive to encourage clients to resolve dependency issues.*** We found in RQ1 that GKIRs can stay open for a median of 6 days, even when the offending dependency is a runtime dependency. This is an especially long time to resolve an issue that is potentially affecting the users of the client project, preventing them from successfully building and installing the client's project in their own project, resulting in more failed builds.

A more effective approach may be to provide telemetry data on the IR itself. For example, the bot could make use of GitHub's dependency graph mechanism[49] to determine the dependents of the client's package and monitor the publicly available data of build systems (e.g., TravisCI[50]) of these dependencies. The bot could then report the number of failed build attempts that have occurred since the IR had been opened. Additionally, this sort of telemetry data could provide a reasonable indicator of whether the IR is a false alarm. We found in RQ2 that GKIRs are often false alarms, usually caused by the client's CI pipeline. Providing the telemetry data on the number of build attempts of the client package in production could give a clearer picture of whether the dependency update has really broken the client's project and how widespread the issue is amongst the client's users. This information could help client developers quickly filter out false alarms, allowing them to react to these issues faster.

Implication 5) ***Dependency management bots should take the type of CI failure into consideration when creating issues for new dependency releases.*** While we recommend in Implication 2 that bots should take into account the initial state of the client's main branch when attempting to test dependency updates, they should also take into account the reason the CI pipeline fails when testing a new dependency update on an isolated branch. Rather than treating all CI pipeline failures the same, these bots should be able to distinguish between CI pipelines that failed because of valid issues potentially caused by the dependency update and issues that are obviously unrelated to the dependency update. After all, these are dependency management bots, which clients expect to use to manage their dependencies, not to use as an alerting mechanism for when something in general is wrong with their CI pipeline.

For example, Gallaba and McIntosh [26] describe tools in their study on misuse of CI features that automatically detect and remove semantic violations in Travis CI build configuration files. Dependency management bots could employ similar techniques to automatically classify common CI failure types.

[49]https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph
[50]https://www.travis-ci.com/

5.2 State-of-the-Art & Future Work in Automated Dependency Management

While Greenkeeper was a very popular dependency management bot, creating over 130,000 PRs [55] and having been referenced in multiple studies [11, 38, 55], it has since been acquired by Snyk[51] and deactivated on June 3 2020, and as such is no longer available for clients to integrate with on GitHub. However, Greenkeeper was one of the first dependency management bots available for use by software developers, and it is likely the designs of dependency management bots that followed were influenced by Greenkeeper.

So, even though Greenkeeper is not state-of-the-art, it has many common features that have been implemented in current state-of-the-art dependency management bots. Still, newer bots may have implemented additional features that can help to reduce the overhead they introduce on client developers. Therefore, we explore and discuss features from three other popular state-of-the-art dependency management bots available for open source software developers: Dependabot[52], Renovate[53], and Depfu[54]. We select these 3 bots to discuss (in addition to Greenkeeper) as they are actively available across multiple ecosystems and have created the most PRs on GitHub of all dependency management bot accounts [55]. We discuss common features available from all of these bots, as well as unique features from each that aim to help ease the overhead they introduce on client developers and how these state-of-the-art bots square with our aforementioned implications.

All of the default configurations of the aforementioned dependency management bots (including Greenkeeper) perform essentially the same task: when one of a client's dependencies releases a new version, the bot will create a new branch with the new version applied, run the client's CI pipeline, and notify the client of the results with the option to update their dependency specifications.

Each of these bots can be configured with multiple options, including ways that can help decrease the amount of overhead they introduce on client developers. For example, clients can set the bot to ignore certain dependencies if they know they are never going to update said dependency and don't want to be bothered by these notifications from the bot.

Greenkeeper, Renovate, and Depfu all offer the option to bundle dependency updates in order to reduce the amount of notifications received by client developers, which is one of our recommendations in Implication 1. It is notable that Dependabot, which is currently the most popular automated dependency management tool available, does not currently support this feature, even though it is a highly requested feature.[55]

Dependabot, Renovate, and Depfu all offer the option for clients to configure how often and at what date/time the bot will attempt to update the client's dependencies. This is a useful feature that can save client developers from a flood of notifications if one of their dependencies frequently releases updates, as the bot will only open a single PR to update the provider to the latest release at the scheduled time. These three bots can also be configured to only open a maximum number of concurrent PRs in the client project, so that client developers do not become overwhelmed with dependency updates. Additionally, Renovate and Depfu can both be configured to automatically merge dependency updates if the client's CI pipeline passes, which is in-line with Implication 1, and can help to further reduce the number of concurrently opened PRs in client projects.

While there are many similarities between these bots, they each have unique features that attempt to minimize the amount of noise they introduce in client projects, and therefore reduce the overhead that comes with integrating with

[51]https://snyk.io/
[52]https://github.com/dependabot
[53]https://github.com/renovatebot/renovate
[54]https://depfu.com/
[55]https://github.com/dependabot/dependabot-core/issues/1190

these bots. Greenkeeper will remain silent on in-range updates that pass the client's CI pipeline. For example, if a client specifies a dependency constraint as "`P: ^1.0.0`", and the provider package releases version `1.0.1`, Greenkeeper will run the client's CI pipeline with the new provider version applied, and remain silent if the pipeline runs successfully (a GKIR will be created if the pipeline fails). Other bots (e.g., Dependabot) will still run the client's CI pipeline with the new provider version applied, but will default to creating a PR to bump the client's version specifications to "`P: ^1.0.1`" (e.g., Dependabot) or "`P: 1.0.1`" (e.g., Renovate), regardless of the outcome of the client's CI pipeline. The implemented behavior by Greenkeeper aims to reduce the number of notifications received by clients[56], and encourages developers to use version range statements (rather than specific versions) for their dependency specifications[57]. Dependabot can be configured to act in a similar manner using its `versioning-strategy`[58] option, where clients can specify how Dependabot should modify the dependency specification file when updating dependencies. For example, client's can specify the `widen` strategy, where Dependabot will relax the version requirement to include both the new and old version, when possible, or the `increase-if-necessary` strategy, where Dependabot will increase the version requirement only when required by the new version.

Renovate specifically allows clients to configure the types of notifications they would like to ignore using the `suppressNotifications`[59] option. For example, clients can disable notifications from a PR being closed without being merged, or can choose to not receive a warning notification for deprecated dependency releases. Renovate also includes a `stabilityDays`[60] option in which clients can configure the number of days required before a new release is considered to be stabilized. This feature is intended to help protect client developers from accepting provider releases that later become unpublished (e.g., npm packages less than 3 days old can be unpublished, which could result in a service impact if the client has already updated to it). These are helpful features that can help to address the amount of activity created by dependency management bots (Implication 1), although it is unclear how often clients actually make use of them in practice.

Depfu has an update strategy called "reasonably up-to-date"[61] that clients can use to reduce the amount of activity generated by the bot. The rationale behind this update strategy is that there is a lot of value in a client having their dependencies up-to-date, but there is very little value in being on all the latest versions. In other words, clients just want their dependencies to stay current. When enabled, Depfu will let new provider releases "mature" before creating a PR in the client project, reducing the amount of dependency updates that clients receive, especially if the provider package has a high release frequency. In fact, the developers of the Depfu bot have tested this feature and found that clients can see a reduction of up to 50% in the amount of PRs opened by the bot[62], significantly reducing the amount of noise clients must deal with (Implication 1.

While state-of-the-art dependency management bots have begun to implement features specifically tailored to help reduce the amount of overhead introduced on the client, there is still room for improvement. For example, none of the aforementioned bots have implemented advanced features, such as examining the logs of the client's CI pipeline to determine the root cause of a failure if a dependency update does not pass the client's CI pipeline (Implication 5), in order to help catch false positive breaking updates. Additionally, these bots do not provide any form of incentive

[56]https://github.com/greenkeeperio/greenkeeper/issues/990
[57]https://github.com/greenkeeperio/greenkeeper/issues/247
[58]https://docs.github.com/en/code-security/supply-chain-security/keeping-your-dependencies-updated-automatically/configuration-options-for-dependency-updates#versioning-strategy
[59]https://docs.renovatebot.com/configuration-options/#suppressnotifications
[60]https://docs.renovatebot.com/configuration-options/#stabilitydays
[61]https://depfu.com/blog/reasonably-up-to-date
[62]https://depfu.com/blog/reasonably-up-to-date

to resolve breaking dependency updates (Implication 4), which may lead to client developers simply ignoring the recommendations of the bot. These features represent interesting avenues for future researchers to study, both in terms of the practicality and efficacy of implementing these features aimed at reducing the overhead introduced on client developers, in addition to the features already implemented by these bots.

Another interesting avenue for future research is exploring the efficacy of automated dependency management bots employing sophisticated methods to automatically detect locations in the client's code affected by non-backwards compatible provider updates (such as in Møller et al. [41]), and transform those parts of the code to become compatible with the new provider version (such as in Nielsen et al. [44]). Combining these areas of research could prove to be an effective approach to reducing the effort on the part of client developers to address GKIRs or, more generally, breaking changes. Additionally, such methods could help to reduce the amount of noise generated by automated dependency bots, as these tools could help to identify when, for example, Greenkeeper has created a GKIR, but the code analysis tools have not detected any non-backwards compatible API changes in the provider package, and can flag the GKIR as a potential false alarm. However, as current state-of-the-art automated dependency management bots do not presently employ such sophisticated features and our study looks to evaluate the overhead introduced by these bots in today's software systems, we leave this to future work.

Also, further research should be done to explore the true amount of work that is required to address breaking dependency issues. While we explore the overhead that is introduced by GKIRs using metrics such as the time to resolve the issue and the size of changes required to resolve the issue, there are additional, more implicit factors (e.g., debugging) that affects the true effort required on the part of client developers to resolve GKIRs. However, this is not always easily measured, as it is difficult to accurately quantify the amount of work required on the developers part to address GKIRs, or software issues in general, and the time needed to resolve issue reports may not always correlate with the actual effort needed to resolve issue reports [35]. For example, on the one hand, an issue report may require minimal effort to resolve but have a low priority, and therefore remains open for an extended period of time, as client developers may delay addressing the issue if they are already overloaded with work. On the other hand, even if only a small change was required to resolve an issue report, the client developer may have expended a significant amount of effort debugging the issue to determine exactly where the issue occurs and exactly what section of the code needed to be modified to resolve the issue.

6 THREATS TO VALIDITY

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

Internal Validity: Threats to internal validity concerns factors that could have influenced our analysis and findings. When we were looking for projects with GKIRs, we only searched for IRs using the GitHub Search API that match the default title used by Greenkeeper. Clients are able to configure the default title that Greenkeeper will use when opening GKIRs, so we may have missed projects that have integrated with Greenkeeper that do not use the default title for their GKIRs. However, we only found 19 projects in our dataset where the client had switched from using the default Greenkeeper title to a custom title, so we do not believe this scenario is very common.

When parsing the information from GKIRs, we were not able to successfully extract the provider dependency type from approximately 4% of GKIRs. While this is still a relatively high success ratio, we only use this information for a single angle of our study in RQ1, so omitting these cases would not have had a major affect on our analysis.

In RQ1, we compare the time taken to close GKIRs with non-GKIRs to explore whether GKIRs are resolved at a faster pace. However, the time taken to close IRs in general can be influenced by many factors (e.g., project maintainers simply might not have enough time to fix issues quickly). We attempt to mitigate this thread by comparing GKIRs and non-GKIRs at the project level, so that project-level factors will be accounted for in our analysis.

Also, in RQ1 the first author manually analyzed comments left by users on GKIRs to extract common patterns so that they could be grouped into similar categories. Only patterns that matched at least 1% of the comments were used, so comments that did not follow a common pattern may not have been matched to a specific category, and therefore the percentage of comments classified to each category represent a lower bound. However, nearly 80% of the total comments were able to be classified, which is a reasonably high proportion.

We cannot definitively provide an exact proportion of GKIRs that are considered noise, as we would have to conclude whether each GKIR was in fact created as a result of a dependency update causing the client's CI pipeline to fail. However, we are able to provide a lower bound of 1.8% using specific comments left from developers on GKIRs (Observation 5), as well as a general approximation of 54.5% for the proportion of GKIRs that are considered noise if we consider all GKIRs with a failed pin attempt to be noise (Observation 6).

In RQ2 we manually analyze a sample of GKIRs that have a failed pin attempt to identify the reasons for why these pin attempts fail. This analysis is subject to author bias, as every investigator has a subjective method when classifying an error that leads to a failed CI pipeline. We mitigate this threat by having the first and second authors independently classify the reasons for CI pipelines failing on 15% of the random samples, then calculating the inter-rater agreement in our methodology (Cohen's Kappa coefficient [14]), after which categories were consolidated as necessary. The level of agreement (+0.93) indicates that the classification results made by the first author are more likely to hold [31, 49], and that the first author could independently classify the remaining 85% of the samples using the agreed upon categories, which is a common approach and has been done in previous work [20, 21, 36].

When collecting commits from client projects to evaluate the level of maintenance activity required to resolve GKIRs, we look for referenced issue events on GKIRs that include a *commit_sha* attribute, which indicates a relationship between the GKIR and the associated commit. However, in order for a commit issue event to be created for a GKIR, the client would have to reference the GKIR issue number either on the commit message when the commit was created, or on the PR that the commit was merged in. While these two heuristics are the main ones used in practice, not every client project may follow these processes.

External Validity: Threats to external validity concern the generalization of our technique and findings. Our study analyses GKIRs opened by Greenkeeper during the period from October 10, 2016 to June 3, 2020. As previously mentioned, while Greenkeeper was a very popular dependency management bot during this time period, creating over 130,000 pull requests [55] and having been referenced in multiple studies [11, 38, 55], it has since been acquired by Snyk[63] and deactivated on June 3 2020, and as such is no longer available for clients to integrate with on GitHub. While we considered including data from Snyk, which offers a similar service, there are differences between the two bots that might lead to inconsistencies in the analysis. Also, there are other dependency management bots in addition to Greenkeeper and Snyk, such as Dependabot, Renovate, and Depfu[64] that should be studied in future work, as they all have unique features that might affect the generalizability of our results with Greenkeeper. However, Greenkeeper was one of the first dependency management bots available for use by software developers, first being released at least a year before the aforementioned dependency management bots, and it is likely the designs of the dependency

[63]https://snyk.io/
[64]https://depfu.com/

management bots that followed were influenced by Greenkeeper. So, while our results cannot be generalized, our discussion provides implications that can still apply to these bots.

Because the collected Greenkeeper data is exclusively from npm, our findings might not be generalizable to other ecosystems. Although npm is representative in size, each software ecosystem has its own intrinsic characteristics, such as the frequency of package releases, the automatic update mechanism, and how package changes are communicated across the ecosystem [7]. Therefore, we acknowledge that additional studies are required in order to further generalize our results. However, to the best of our knowledge, this is the first paper to empirically analyze the potential overhead that is introduced by dependency management bots and provide a series of practical recommendations for designers of these bots.

7 CONCLUSION

It has become commonplace for developers to reuse code from multiple provider packages in the form of software dependencies. With this rise in software dependencies in open source software projects, we have seen an increase in popularity of using software bots to automatically manage these dependencies. Although bots are able to help automate these monotonous tasks, integrating these bots into a project's workflow introduces a certain level of overhead in the client project, and once the bot begins performing its specific function, human intervention is usually required to either accept or reject any actions or recommendations the bot creates.

In this paper, we perform an empirical study of 93,196 issue reports opened by Greenkeeper (GKIRs), a popular software bot used to manage software dependencies in the npm ecosystem, that examines the extent to which automated dependency management bots can either save or create unnecessary work in their client projects. Studying these GKIRs allows us to explore the amount of overhead created by using these types of dependency management bots. Specifically, we examine the overhead introduced in client projects by Greenkeeper (RQ1). Our results show that Greenkeeper introduces a significant amount of overhead in the form of notifications and other artifacts (e.g., issue reports and comments) that must be addressed by client developers. Next, we explore whether automated dependency pinning is an effective mechanism for resolving GKIRs (RQ2), and observe that this is not the case, with 68% of pin attempts failing, usually due to reasons unrelated to the dependency update (e.g., pre-existing issue in the client's CI pipeline). Finally, we look at the performed code changes resolving GKIRs (RQ3). We observe that, while the majority of changes that resolve GKIRs are small (1-3 LOC) modifications to the client's dependency specification file, they can sometimes require changes to the client's source code, in which case they are comparable in size to changes that resolve non-GKIRs.

These findings indicate that, while bots like Greenkeeper can be an effective tool for managing dependencies, they also can generate a significant amount of noise in client projects, especially if the client has a low quality CI pipeline that is prone to intermittent failures. Leveraging our findings, we provide a series of implications that are of interest for designers of dependency management bots, with attention given to practical recommendations to help reduce the amount of overhead introduced by these bots.

Acknowledgements The findings and opinions in this paper belong solely to the authors, and are not necessarily those of Huawei. Moreover, our results do not in any way reflect the quality of Huawei software products.

REFERENCES

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 385–395. https://doi.org/10.1145/3106237.3106267

[2] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empirical Software Engineering* 25, 2 (March 2020), 1168–1204. https://doi.org/10.1007/s10664-019-09792-9

[3] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the Use of Dependabot Security Pull Requests. In *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories*. IEEE, Madrid, Spain, 254–265. https://doi.org/10.1109/MSR52588.2021.00037

[4] Richard A. Armstrong. 2014. When to use the Bonferroni correction. *Ophthalmic and Physiological Optics* 34, 5 (Sept. 2014), 502–508. https://doi.org/10.1111/opo.12131

[5] David F. Bauer. 1972. Constructing Confidence Sets Using Rank Statistics. *J. Amer. Statist. Assoc.* 67, 339 (Sept. 1972), 687–690. https://doi.org/10.1080/01621459.1972.10481279

[6] Christopher Bogart, Christian Kastner, and James Herbsleb. 2015. When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop*. IEEE, Lincoln, NE, 86–89. https://doi.org/10.1109/ASEW.2015.21

[7] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Seattle WA USA, 109–120. https://doi.org/10.1145/2950290.2950325

[8] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Oct. 2016), 334–344. https://doi.org/10.1109/ICSME.2016.31 arXiv: 1606.04984.

[9] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Campobasso, 507–511. https://doi.org/10.1109/SANER.2018.8330249

[10] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Campobasso, 255–265. https://doi.org/10.1109/SANER.2018.8330214

[11] Chris Brown and Chris Parnin. 2020. Sorry to Bother You Again: Developer Recommendation Choice Architectures for Designing Effective Bots. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ACM, Seoul Republic of Korea, 56–60. https://doi.org/10.1145/3387940.3391506

[12] Norman Cliff. 1996. *Ordinal methods for behavioral data analysis.* Lawrence Erlbaum Associates, Inc, Hillsdale, NJ, US. Pages: xiii, 197.

[13] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E. Hassan. 2019. An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2952130 Conference Name: IEEE Transactions on Software Engineering.

[14] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (April 1960), 37–46. https://doi.org/10.1177/001316446002000104

[15] Joel Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 109–118. https://doi.org/10.1109/ICSE.2015.140

[16] Alexandre Decan and Tom Mens. 2020. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2019.2918315

[17] Alexandre Decan, Tom Mens, and Maelick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Klagenfurt, Austria, 2–12. https://doi.org/10.1109/SANER.2017.7884604

[18] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Madrid, 404–414. https://doi.org/10.1109/ICSME.2018.00050

[19] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories*. IEEE, Montreal, QC, Canada, 349–359. https://doi.org/10.1109/MSR.2019.00061

[20] Marcia W. DiStaso and Denise Sevick Bortree. 2012. Multi-method analysis of transparency in social media practices: Survey, interviews and content analysis. *Public Relations Review* 38, 3 (Sept. 2012), 511–514. https://doi.org/10.1016/j.pubrev.2012.01.003

[21] Margaret Drouhard, Nan-Chen Chen, Jina Suh, Rafal Kocielnik, Vanessa Pena-Araya, Keting Cen, Xiangyi Zheng, and Cecilia R. Aragon. 2017. Aeonium: Visual analytics to support collaborative qualitative coding. In *Proceedings of the 2017 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, Seoul, South Korea, 220–229. https://doi.org/10.1109/PACIFICVIS.2017.8031598

[22] Linda Erlenhov, Francisco Gomes de Oliveira Neto, Riccardo Scandariato, and Philipp Leitner. 2019. Current and Future Bots in Software Development. In *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering*. IEEE, Montreal, QC, Canada, 7–11. https://doi.org/10.1109/BotSE.2019.00009

[23] Amin Milani Fard and Ali Mesbah. 2017. JavaScript: The (Un)Covered Parts. In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Tokyo, Japan, 230–240. https://doi.org/10.1109/ICST.2017.28

[24] Joseph L. Fleiss and Jacob Cohen. 1973. The Equivalence of Weighted Kappa and the Intraclass Correlation Coefficient as Measures of Reliability. *Educational and Psychological Measurement* 33, 3 (Oct. 1973), 613–619. https://doi.org/10.1177/001316447303300309

[25] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista FL USA, 791–796. https://doi.org/10.1145/3236024.3275535

[26] Keheliya Gallaba and Shane McIntosh. 2020. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (Jan. 2020), 33–50. https://doi.org/10.1109/TSE.2018.2838131

[27] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *Open Source Systems: Towards Robust Practices*. Vol. 496. Springer International Publishing, 182–192. https://doi.org/10.1007/978-3-319-57735-7_17 Series Title: IFIP Advances in Information and Communication Technology.

[28] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2020. Dependency Smells in JavaScript Projects. *arXiv:2010.14573 [cs]* (Oct. 2020). http://arxiv.org/abs/2010.14573 arXiv: 2010.14573.

[29] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break – An empirical study. *Information and Software Technology* 65 (Sept. 2015), 129–146. https://doi.org/10.1016/j.infsof.2015.02.014

[30] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration. *Empirical Software Engineering* 23, 1 (Feb. 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5 arXiv: 1709.04621.

[31] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (March 1977), 159. https://doi.org/10.2307/2529310

[32] Carlene Lebeuf, Alexey Zagalsky, Matthieu Foucault, and Margaret-Anne Storey. 2019. Defining and Classifying Software Bots: A Faceted Taxonomy. In *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering*. IEEE, Montreal, QC, Canada, 1–6. https://doi.org/10.1109/BotSE.2019.00008

[33] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: automating the detection of API-related compatibility issues in Android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Amsterdam Netherlands, 153–163. https://doi.org/10.1145/3213846.3213857

[34] Bin Lin, Alexey E. Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. 2016. Why Developers Are Slacking Off: Understanding How Software Teams Use Slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion - CSCW '16 Companion*. ACM Press, San Francisco, California, USA, 333–336. https://doi.org/10.1145/2818052.2869117

[35] Lionel Marks, Ying Zou, and Ahmed E. Hassan. 2011. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (Promise '11)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2020390.2020401

[36] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–23. https://doi.org/10.1145/3359174

[37] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *Proceedings of the 32nd European Conference on Object-Oriented Programming*. 24 pages. https://doi.org/10.4230/LIPICS.ECOOP.2018.7 Artwork Size: 24 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

[38] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Urbana, IL, 84–94. https://doi.org/10.1109/ASE.2017.8115621

[39] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using Others' Tests to Identify Breaking Updates. In *Proceedings of the 17th International Conference on Mining Software Repositories*. ACM, Seoul Republic of Korea, 466–476. https://doi.org/10.1145/3379597.3387476

[40] J.C. Munson and S.G. Elbaum. 1998. Code churn: a measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE Comput. Soc, Bethesda, MD, USA, 24–31. https://doi.org/10.1109/ICSM.1998.738486

[41] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–25. https://doi.org/10.1145/3428255

[42] Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node.js libraries. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM Press, Tallinn, Estonia, 409–419. https://doi.org/10.1145/3338906.3338940

[43] N. Nagappan and T. Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 2008 Frontiers of Software Maintenance, FoSM 200827th International Conference on Software Engineering, 2005. ICSE 2005*. ACM, 284–292. https://doi.org/10.1109/ICSE.2005.1553571 ISSN: 1558-1225.

[44] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries. *Proc. 43rd International Conference on Software Engineering (ICSE)* (2021), 12.

[45] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, Victoria, BC, Canada, 215–224. https://doi.org/10.1109/SCAM.2014.30

[46] S. Raemaekers, A. van Deursen, and J. Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (July 2017), 140–158. https://doi.org/10.1016/j.jss.2016.04.008

[47] Jeanine Romano and Jeffrey Kromrey. 2006. Appropriate Statistics for Ordinal Level Data: Should We Really Be Using t-test and Cohen's d for Evaluating Group Differences on the NSSE and other Surveys? (2006).

[48] Benjamin Rombaut, Filipe R. Cogo, Bram Adams, and Ahmed E. Hassan. 2022. Greenkeeper Overhead - Online Appendix. https://github.com/SAILResearch/suppmaterial-22-ben-greenkeeper-overhead

[49] Julius Sim and Chris C Wright. 2005. The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements. *Physical Therapy* 85, 3 (March 2005), 257–268. https://doi.org/10.1093/ptj/85.3.257

[50] Margaret-Anne Storey and Alexey Zagalsky. 2016. Disrupting developer productivity one bot at a time. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. ACM Press, Seattle, WA, USA, 928–931. https://doi.org/10.1145/2950290.2983989

[51] Mairieli Wessel. 2020. Enhancing developers' support on pull requests activities with software bots. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 1674–1677. https://doi.org/10.1145/3368089.3418539

[52] Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A. Gerosa. 2018. The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Proceedings of the ACM on Human-Computuer Interaction* 2, 182 (Nov. 2018), 19. https://doi.org/10.1145/3274451

[53] Mairieli Wessel and Igor Steinmacher. 2020. The Inconvenient Side of Software Bots on Pull Requests. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ACM, Seoul Republic of Korea, 51–55. https://doi.org/10.1145/3387940.3391504

[54] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, Austin Texas, 351–361. https://doi.org/10.1145/2901739.2901743

[55] Marvin Wyrich, Raoul Ghit, Tobias Haller, and Christian Müller. 2021. Bots Don't Mind Waiting, Do They? Comparing the Interaction With Automatically and Manually Created Pull Requests. *arXiv:2103.03591 [cs]* (March 2021). http://arxiv.org/abs/2103.03591 arXiv: 2103.03591.

[56] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An Empirical Analysis of Technical Lag in npm Package Dependencies. In *New Opportunities for Software Reuse*. Vol. 10826. Springer International Publishing, 95–110. https://doi.org/10.1007/978-3-319-90421-4_6 Series Title: Lecture Notes in Computer Science.