Studying the overhead and crowd-sourced risk assessment strategy of Dependency Management Bots

by

Benjamin John Rombaut

A thesis submitted to the School of Computing

in conformity with the requirements for the

Degree of Master of Science

Queen's University

Kingston, Ontario, Canada

May 2022

Abstract

As today's software systems are increasingly built with dependency relationships, where a client package makes use of a specific version of a provider package, these client packages must effectively manage their dependencies. To help facilitate this dependency management process, clients are increasingly adopting dependency management bots to alert them when a provider package they depend on releases a new version and whether the new version of said provider package is compatible with their package.

Integrating these dependency management bots into a project requires a certain level of effort on the part of the client, 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 generates. This creates additional, and sometimes unnecessary, work for clients, which can deter them from continuing to use the bot. Additionally, dependency management bots have begun to implement the promising strategy of leveraging “the crowd” to help clients assess the involved risks with accepting a dependency update. This opportunity to use knowledge from “the crowd” to aid client packages with dependency management is interesting and unique to dependency management bots, as they have access to the vast store of data representing how compatible each provider package release is across many client packages. In this thesis, we present two studies that examine these attributes of dependency management bots. First, we describe a large empirical study on the overhead that is introduced in client packages that adopt dependency management bots. In particular, we provide a series of practical recommendations to help designers of dependency management bots reduce the amount of unnecessary work they create in client packages. Next, we describe a large scale study on the efficacy of dependency management bots leveraging “the crowd” to provide supporting metrics to help clients assess the risk of accepting a dependency update. Our findings will help designers of dependency management bots effectively leverage crowd-sourced data to aid client packages with dependency management.

Acknowledgments

First, thanks so much to Dr. Filipe R. Cogo. Filipe, I'm very lucky to have worked with you and sincerely appreciate all the help and guidance you've given throughout my degree. I'm looking forward to continuing to work with you in the future.

Special thanks to my supervisor Dr. Ahmed E. Hassan for his guidance and allowing me to pursue interesting research at SAIL. You've made the lab a great place to work, and I wish all other SAILors the best of luck.

Also, thanks to Dr. Bram Adams and Dr. Gustavo A. Oliva. Bram, thanks for all your valuable feedback as both an instructor and a collaborator. Gustavo, thanks for your advice and all the work you do for the lab.

Thanks to everyone with the Software Engineering program at the University of New Brunswick, especially Dr. Dawn MacIsaac. Software Engineering students are very lucky for all the work you do to support such an excellent program. My time at

UNB was amazing, and the support of numerous friends and the great experiences that I had have helped in furthering both my academic and professional career.

Nick - thanks for making grad school during Covid bearable.

Finally, Mom, Dad, Josh & Sis - thanks for everything.

Co-authorship

For each of the chapters and related publications of this thesis, my contributions are: the drafting of the research idea; researching of the background material and related work; the collection of the data; the proposal of the research methods; the analysis of the data; and the writing of the manuscripts. My co-contributors supported me in refining the initial research ideas; providing suggestions to refine my research methods; and providing feedback on manuscript drafts. The work presented in this thesis is submitted as listed below:

- 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 Transactions on Software Engineering and Methodology (TOSEM). Accepted. This work is described in Chapter 4.

- Benjamin Rombaut, Filipe R. Cogo, and Ahmed E. Hassan. 2022. Leveraging the Crowd for Dependency Management: An Empirical Study on the Dependabot Compatibility Score. IEEE Transactions on Software Engineering (TSE). Under review. This work is described in Chapter 5.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

TODAY'S software systems are rarely built from scratch, with client packages often making use of specific versions of provider packages through 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 (Lim, 1994; Basili et al., 1996; Mohagheghi et al., 2004).

However, clients must also incur the cost of managing these dependencies, as provider packages continuously release new versions containing bug fixes, new functionalities, and security enhancements. Still, clients cannot blindly accept every dependency update, as there is an inherent risk that these new provider versions may modify existing functionality or introducing backwards incompatibilities (a.k.a., breaking updates).

To help facilitate the process of dependency management, clients are increasingly adopting dependency management bots to alert them when a provider on which they depend releases a new version, and whether the new version of the provider is compatible with their package (Storey and Zagalsky, 2016; Wessel et al., 2018). Integrating these bots into a project's workflow requires a certain level of effort on the part of the client, and once the bot begins performing its specific function, human intervention is usually required to either accept or reject any action or recommendation by the bot. Such bot recommendations can create additional, and sometimes unnecessary, work for clients, which can deter them from continuing to use the bot.

Perhaps the most common action generated by these dependency bots on which clients must act is whether the client should accept a recommendation of a dependency update by the bot. To support clients in making this decision, bots may provide additional metrics on the new release of the provider, such as how compatible said release is with other client packages that also make use of the same provider as a dependency. This opportunity to use knowledge from "the crowd" to help clients assess the involved risk with a dependency update is interesting and unique to dependency management bots, as they have access to a vast store of data representing how compatible each provider package release is across many client packages.

In this thesis, we study the overhead that is introduced in clients that adopt these dependency management bots, as well as explore the viability of dependency management bots leveraging "the crowd" to help clients assess the involved risks with accepting a dependency update. Based on our findings, we provide a series of implications that are of interest for the designers of dependency management bots, with attention given to practical recommendations to help reduce the amount of overhead introduced by these bots, as well as how bots can effectively leverage the crowd to further aid clients with dependency management.

1.1 Thesis Statement

Client packages are increasingly turning to bots to help facilitate the essential and risky task of dependency management. Therefore, it is important for designers of these dependency management bots to understand issues clients face when adopting these bots, as well as how to effectively provide supporting metrics that aid clients with dependency management.

1.2 Thesis Overview

In this section, we provide an outline of our thesis.

1.2.1 Chapter 2: Background

In this chapter, we provide background information related to dependency management, discussing how practitioners manage their dependencies and common issues related to dependency management in software ecosystems. We then provide background information on software bots in general and how dependency management bots are used to help alleviate the required work of practitioners in regard to dependency management.

1.2.2 Chapter 3: Related Work

In this chapter, we provide an overview of prior research that is related to different aspects of dependency management, including the prevalence of dependency usage in software ecosystems, whether clients keep their dependencies up-to-date, common issues related to dependency management, and methods that clients employ to guard against breaking dependency updates. We then discuss existing work related to how bots are used in software engineering, the effects of adopting bots in software projects and how bots are perceived by human developers, and bots specifically designed for dependency management.

1.2.3 Chapter 4: Exploring the overhead introduced by the Greenkeeper dependency bot

As it becomes more commonplace for client packages to make use of software dependencies, we have also 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 package's workflow introduces a certain level of overhead for the client package, and once the bot begins performing its specific function, human intervention is usually required to either accept or reject any actions or recommendations by the bot.

In this chapter, we describe an empirical study on the overhead that is incurred by clients as a result of adopting the Greenkeeper[1] dependency management bot. We examine Greenkeeper issue reports (GKIRs), which are issue reports opened by Greenkeeper when the bot detects a dependency-related problem in client packages, to explore the extent to which dependency management bots can either save or create unnecessary work for these clients. In particular, we study (i) the overhead introduced in client packages by the bot, (ii) the efficacy of recommended actions by the bot, and (iii) the size of manual changes that are required by client developers to resolve issues created by the bot. We observe that the bot introduces a significant amount of work in the form of notifications and other artifacts (e.g., issue reports and comments) that must be addressed by client developers. We also observe that automatically attempting to downgrade the failing dependency to the previous working version (i.e., *pinning*), which should reverse any breaking changes introduced by the dependency update with the least amount of effort and is automatically attempted by Greenkeeper, is not an effective mechanism for resolving issues that are created by the bot, as it fails more than two-thirds of the time. 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. Finally, we observe that, while the majority of changes that resolve GKIRs are small (1-3 lines of code) modifications to the client's dependency

[1] https://greenkeeper.io/

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. 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.

1.2.4 Chapter 5: On Bots Leveraging the Crowd for Dependency Management: An Empirical Study of the Dependabot Compatibility Score

As the prevalence of client packages making use of provider packages in the form of dependency relationships increases, client packages must face the essential and risky task of keeping their provider packages up-to-date. To help facilitate this task, clients are increasingly adopting dependency management bots to automatically update and test a new version of a provider package when it is released. This presents a unique opportunity, in that these bots have access to a vast store of data representing how compatible each provider package release is across many client packages.

Dependabot[2], perhaps the most popular dependency management bot, is the first to take advantage of this opportunity by providing a *compatibility score* for each provider package release. This compatibility score is shown as a badge on

[2] https://github.com/dependabot

pull requests (PRs) opened by Dependabot, and is meant to give clients a sense of the involved risk when updating a provider package by leveraging the knowledge of "the crowd", so that clients can be confident that a new provider version is backwards compatible and bug-free.

In this chapter, we describe an empirical study on the efficacy of Dependabot leveraging the crowd to provide a compatibility score to help clients assess the risk of accepting a dependency update. Specifically, this chapter includes (i) an empirical study that examines how effective Dependabot's current compatibility score strategy, (ii) a description and evaluation of additional data sources that dependency management bots can consider when the crowd does not provide enough support to calculate a compatibility score, and (iii) a description and evaluation of an approach dependency management bots can employ to help calibrate the level of trust clients should place in compatibility scores. We observe that the majority of compatibility scores do not even have 5 candidate updates, which is the threshold required for the compatibility score badge to be displayed on Dependabot PRs, and when compatibility scores do have enough candidate updates, the vast majority of the scores are above 90%. As a result of this skewness in both the number of candidate updates and the scores themselves, dependency management bots should employ further methods to help amplify the input from the crowd and consider historical upgrade metrics to assess whether a client package should accept or reject a dependency update. Additionally, supporting metrics, such as a confidence interval, should be provided alongside the compatibility score to help calibrate the level of trust client packages should place in the score.

1.3 Thesis Contribution

This thesis is focused towards helping designers of dependency management bots to create tools that are easy to adopt and provide helpful metrics to further aid clients handle the task of dependency management. In particular, our main contributions are as follows:

1. This is the first work to perform a large empirical study on the overhead that is introduced in client packages that adopt dependency management bots. In particular, we provide a series of practical recommendations to help designers of dependency management bots reduce the amount of unnecessary work they create in client packages.

2. This is the first work to study on a large scale the efficacy of dependency management bots leveraging the crowd to provide supporting metrics to help clients assess the risk of accepting a dependency update. Our findings will help designers of dependency management bots effectively leverage crowd-sourced data to aid client packages with dependency management.

CHAPTER 2

Background

IN this chapter, we first provide an overview of how practitioners manage their dependencies and common issues related to dependency management in software ecosystems (Section 2.1). We then discuss how dependency management bots are used to help alleviate the work that is required of practitioners in regard to dependency management (Section 2.2).

2.1 Dependency Management

As more and more client packages depend on provider packages in the form of dependency relationships, and these provider packages continuously release new versions containing new features and improvements, it is important to standardize how these changes are communicated from the provider package to the client package. The most popular policy for communicating the kinds of changes made to a software package is the Semantic Versioning[1] (SemVer) scheme. A SemVer-compatible version is a version number composed of a major, minor and patch number that allows maintainers to logically order package releases. SemVer is adopted, for example, in npm[2][3], the largest software ecosystem, where provider packages need to specify the version number of each release in their *package.json* metadata file[4]. 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.

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., an *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,

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

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 2.1 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 $\geq$1.0.0 $\land$ $<$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 $\geq$1.0.0 $\land$ $<$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.

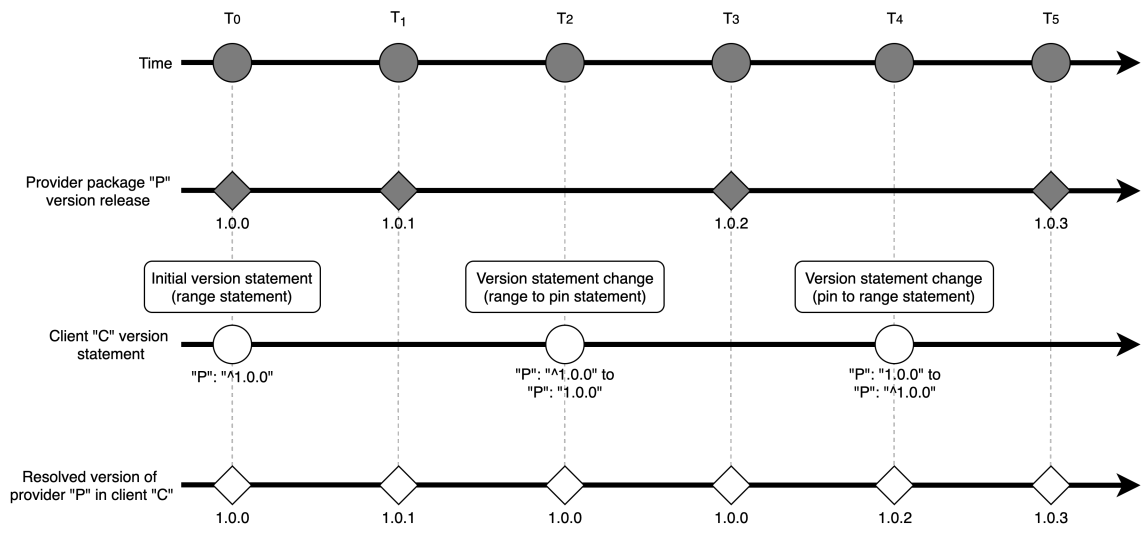

T0
T1
T2
T3
T4
T5
Time
Provider package "P" version release
1.0.0
1.0.1
1.0.2
1.0.3
Initial version statement (range statement)
Version statement change (range to pin statement)
Version statement change (pin to range statement)
Client "C" version statement
"P": "^1.0.0"
"P": "^1.0.0" to "P": "1.0.0"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

Figure 2.1: 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.

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. 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 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.

Client developers can protect themselves from in-range breaking updates by using *lock files* (e.g., `package-lock.json` in npm) 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.[5]

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`, for example, cannot be published to npm[6]. This means that if a user of the client package (e.g., another developer with their own project) installs the client's published package

[5] https://docs.npmjs.com/cli/v8/configuring-npm/package-lock-json

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

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.

If a dependency update breaks a client's build, the client may resort to version pinning their dependencies to resolve the issue. Version pinning a dependency involves a client changing their dependency specification for a provider package 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 2.1. 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. 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 constraints when new versions are released, the client will not receive bug fixes or new functionalities from the provider.

2.2 Software Bots

Lebeuf (2018) defines software bots as:

> *"An interface that connects users to services. These services can be internalized in the bot's code and/or accessed externally. The bot also provides some sort of additional value (in the form of interaction style, automation, anthropomorphism, etc.) on top of the software service's basic capabilities."* - Lebeuf (2018)

In short, bots can be thought of as tools that perform repetitive predefined tasks to save developers' time, increase their productivity, and support them in making smarter decisions (Storey and Zagalsky, 2016). Bots can either run continuously or be triggered by occurrences associated with events, time conditions, or manual execution. Bots can be useful for automatically completing a wide variety of chores, such as automating CI pipelines, detecting flaky tests, and creating issues when a service fails. Consequently, many software developers, teams, and companies take advantage of bots to do these repetitive tasks because bots can perform those tasks more efficiently than human users.

Specifically, client packages 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 related to dependency management 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).

Greenkeeper, Dependabot, and Renovate[7] are popular dependency management bots that clients can integrate into their software packages 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 package. When one of a client's dependencies releases a new version, the bot will create a fresh branch with the new dependency version applied to the client's package, run the client's CI pipeline, and notify the client of the results with the option to update their dependency specifications. The client is then able to either accept or reject the recommendation of the bot.

[7] https://docs.renovatebot.com/

CHAPTER 3

Related Work

In this chapter, we provide an overview of prior research that is related to different aspects of dependency management (Section 3.1), including the prevalence of dependency usage in software ecosystem, whether clients keep their dependencies up-to-date, common issues related to dependency management, and methods that clients employ to guard against breaking dependency updates. We then discuss existing work related to software bots (Section 3.2), including how bots are used in software engineering, the effects of adopting bots in software projects and how bots are perceived by human developers, and bots specifically designed for dependency management.

3.1 Dependency Management

In this section, we discuss existing work related to the prevalence of dependency usage in software ecosystem (Section 3.1.1), whether clients keep their dependencies up-to-date (Section 3.1.2), common issues related to dependency management (Section 3.1.3), and methods clients employ to guard against breaking dependency updates (Section 3.1.4).

3.1.1 Prevalence of Dependency Usage

Multiple studies examine the growing trend of using provider packages to build new software. Wittern et al. (2016) examine the dynamics of npm and find that 81.3% of packages depend on at least one other package, while 32.5% of them depend on 6 or more packages. Fard and Mesbah (2017) find similar results, showing that projects have an average of 6 dependencies and that this number is following a growing trend.

Client packages that make use of a provider package also implicitly adopt all the dependencies that said provider uses as well, called *transitive dependencies*. Kikas et al. (2017) find that the ratio of transitive dependencies to direct dependencies for projects is greater than 10, and packages exist in the ecosystem whose removal could impact up to 30% of the entire ecosystem.

Practitioners will even rely on third-party dependencies to accomplish trivial tasks. Abdalkareem et al. (2017a, 2020) conduct two studies that examine the reasons clients choose to use “trivial” packages in the npm and PyPi ecosystems, and find that clients believe trivial packages provide well-implemented and tested code, as well as increase development productivity. However, they also find that clients worry that trivial packages introduce management overhead as well as a risk of breaking their application. Chowdhury et al. (2021) perform a follow-up study that looks at how trivial these “trivial” packages actually are, and find that trivial packages are often used in central parts of software packages compared to non-trivial packages, and that, while 16.8% of packages in the npm ecosystem are considered trivial, in some cases removing one of them can impact 29% of the ecosystem, further confirming the results obtained by Wittern et al. (2016) and Kikas et al. (2017).

3.1.2 Outdated Dependencies

While clients continue to make use of dependency relationships, they are reluctant to keep their dependencies up-to-date. The term “technical lag”, which was first proposed by Gonzalez-Barahona et al. (2017), refers to packages lagging behind with respect to using the latest version of their dependencies. Decan et al. (2017, 2018a) perform an analysis of multiple package dependency networks on the evolution of technical lag and examined how technical lag can be reduced by relying on the SemVer policy (the efficacy of which has been examined in multiple studies (Raemaekers et al., 2014, 2017; Dietrich et al., 2019; Decan and Mens, 2020; Foo et al., 2018)).

The incurment of technical lag is not the only consequence that results from the decision of clients to not keep their dependencies up-to-date. Decan et al. (2018b) study how clients weigh the risks of using a vulnerable version of a dependency against updating the dependency, and find that more than half of all dependent packages are affected by vulnerabilities in upstream packages. A large fraction of affected dependent packages are not updated, even if an upstream fix is available, usually caused by improper or too restrictive use of dependency constraints and unmaintained packages. Kula et al. (2018) perform an empirical study on over 4,600 GitHub[1] software projects and 2,700 packages to explore the impact of security advisories on dependency updates, and find that 81.5% of the studied systems keep their outdated dependencies regardless of whether there is a security advisory for the dependency, which 69% of the surveyed developers claim to be unaware of. Similarly, Cox et al. (2015) find that systems using outdated dependencies are four times as likely to have security issues as opposed to systems that keep their dependencies up-to-date.

3.1.3 Perils of Dependency Management

While the consequences of clients neglecting to keep their dependencies up-to-date is well known, the obvious question that is raised is why would dependencies not be kept up-to-date? After all, having up-to-date dependencies means that clients benefit from the latest features, bug fixes, and security enhancements as they are released from their dependencies. However, simply accepting the latest version of a dependency is not necessarily a viable option for most packages. There is an inherent risk when a client updates a dependency that the update will break the client's package in some way, and nearly all the studies mentioned thus far cite the risk of breaking changes being one of the primary concerns that clients have when it comes to determining whether they should update their dependencies.

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

Bogart et al. (2015, 2016) discuss their findings from interviews with developers concerning what they feel are primary issues with dependency management and find that developers perceive dependency management and evolution as severe issues and that existing awareness mechanisms (e.g., email notifications) are rarely used. Brito et al. (2018b, 2020) examine the motivations behind developer's decisions to released backwards incompatible changes in their APIs and find that these decisions were mostly motivated by the need to implement new features, to make the package easier to use by simplifying the API, and to improve maintainability.

As explained in Section 2.1, a client may resort to version pinning their dependencies as a workaround to fix breaking updates that occur from a dependency releasing a backwards-incompatible change. Jafari et al. (2020) find that developers choose to pin some of their dependencies in over 52% of npm projects, and Cogo et al. (2019) find 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). Additionally, Zerouali et al. (2018) find that technical lag is often caused by clients using strict dependency version constraints, such as pinning.

3.1.4 Guarding Against Breaking Updates

Multiple studies examine how to detect breaking changes in API updates (Jezek et al., 2015; Brito et al., 2018a; Li et al., 2018). Specifically, Mezzetti et al. (2018) and Møller and Torp (2019) 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.

(2020) 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.

To detect whether clients would be affected by a dependency update, Møller et al. (2020) 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. (2021) 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.

3.2 Software Bots

In this section, we discuss existing work related to the usage of software bots (Section 3.2.1), the effects of adopting bots in software projects and how they are perceived by human developers (Section 3.2.2), and bots for dependency management (Section 3.2.3).

3.2.1 Usage of Software Bots

Storey and Zagalsky (2016) and Lin et al. (2016) find that bots are used to help developers make smarter decisions and to support developers that need to communicate and coordinate with others. Both of these early exploratory studies examine the roles that bots play in different aspects of the software development life-cycle, as well as discuss the challenges and risks that bots may introduce and further research directions. Wessel et al. (2018) show that bots are primarily used for reporting build failures, decreasing code review time, and automating CI pipelines, and that developers want both smarter and more configurable bots to work with.

Erlenhov et al. (2019) and Lebeuf et al. (2019) present taxonomies for classifying software bots based on the observable properties, behaviours, and working environments of different types of bots. They also discuss visions of what they would consider for a future ideal bot, such as being autonomous, adaptive, and technically and socially competent, ultimately acting akin to an artificial teammate rather than a simple development tool.

Finally, multiple studies explore how developers interact with specific bots. For example, Peng et al. (2018) study how developers work with Mention[2] bot, an automatic reviewer recommendation bot for PRs, reporting that developers appreciate the effort that Mention bot saves but are hampered by its unstable setting and unbalanced workload allocation. Urli et al. (2018) examine the efficacy of the automatic program repair bot Repairnator[3], and find that, while the Repairnator allows researchers to collect a unique empirical dataset to study the challenges of program repair, the bot itself has not yet succeeded in proposing an effective patch to human developers. Wessel et al. (2019) analyze the adoption and usage over time of the Stale[4] bot, which helps maintainers in triaging abandoned issues and PRs, and find that developers rarely modify the default configuration file of the bot.

[2]https://github.com/facebookarchive/mention-bot
[3]https://github.com/eclipse/repairnator
[4]https://github.com/marketplace/stale

3.2.2 The Effects of Bots on Software Projects and The Perception by Human Developers

The impact of introducing bots into software projects has been the subject of multiple studies. Wessel et al. (2020a) investigate how several activity indicators change after the adoption of a code review bot, and find that the adoption of a code review bot increases the number of monthly merged PRs, decreases monthly non-merged PRs, and decreases communication among developers. However, Wyrich et al. (2021) find that PRs from bots are only accepted and merged 37% of the time, versus 73% of PRs manually created by human developers. Additionally, they find that it takes significantly longer for a bot PR to be interacted with and for it to be merged, even though they contain fewer changes on average than human PRs.

Numerous studies have examined how software bots are perceived by human developers. After introducing an autonomous refactoring bot into student software development projects, Wyrich et al. (2020) conduct interviews with the students and find that the bot was perceived as a useful and unobtrusive contributor, and that the students were no more critical of it than they were about their human peers.

Similarly, Wessel et al. (2018), Wessel et al. (2020b), Wessel et al. (2021), and Erlenhov et al. (2020) perform interviews with industry practitioners and find that, while bots are used to streamline tasks in software projects, their presence can cause interruptions and noise, trust, and usability issues. Brown and Parnin (2019) also report that bots still need to overcome problems such as notification overload in order to enhance their interactions with humans.

3.2.3 Bots for Dependency Management

Automated dependency management bots have become a popular area of research, being at least partially examined in multiple studies (Erlenhov et al., 2019; Lebeuf et al., 2019; Wessel, 2020; Wessel and Steinmacher, 2020). Most relevant to this thesis, Mirhosseini and Parnin (2017) conduct a study on the effectiveness of different notification techniques designed to help developers keep their dependencies up-to-date. They find 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[5]) upgraded their dependencies 1.6 and 1.4 times as often, respectively, as projects that did not use any tools. Still, they report that developers have negative perceptions of automated PRs and faced challenges convincing users to upgrade out-of-date dependencies with Greenkeeper. While their work specifically looks at whether tools like Greenkeeper can help developers keep their dependencies up-to-date, in the studies described in this thesis we look to measure the degree of unnecessary work that these types of tools create in client projects that use them, as well as whether these types of tools are able to effectively take advantage of their large user base to provide supporting crowd-sourced metrics to help with dependency management.

Also, Alfadel et al. (2021) examine the use of Dependabot for automatically creating PRs to fix dependency vulnerabilities in a client's project. They find 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 find that half of

[5]https://david-dm.org/

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 on the potential overhead introduced by dependency management bots, as well as the efficacy of Dependabot's strategy of leveraging the crowd to provide clients with a sense of the involved risk with accepting a dependency update.

CHAPTER 4

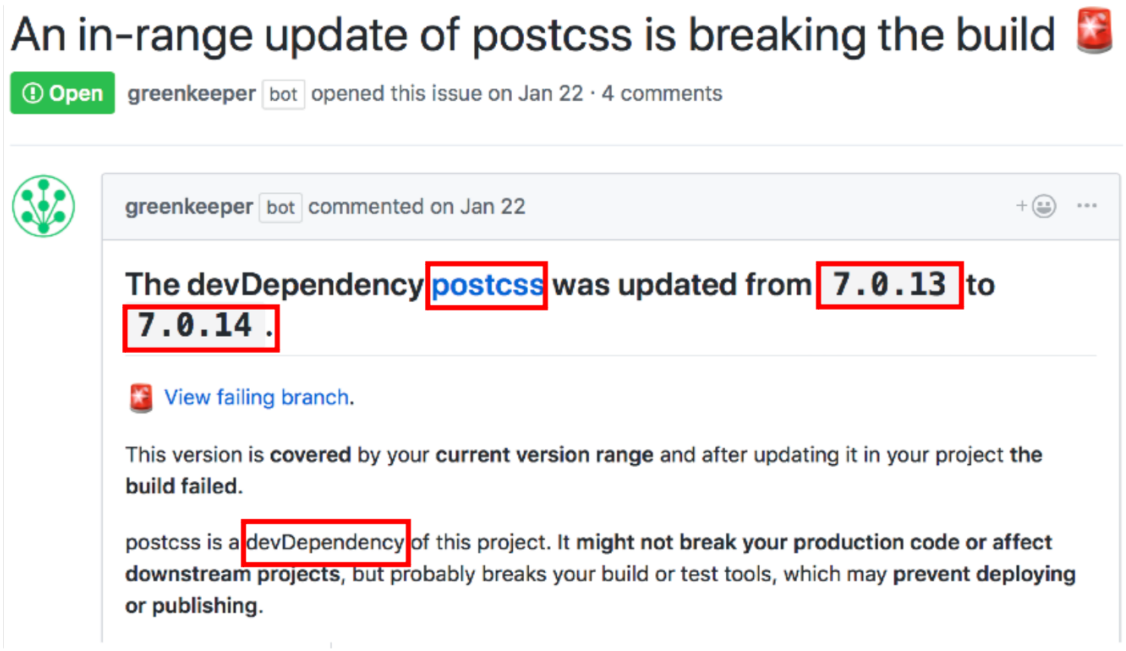
Exploring the overhead introduced by the Greenkeeper dependency bot

This chapter is to appear in an upcoming publication of the ACM Transactions on Software Engineering and Methodology (TOSEM).

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 chapter, we describe our study on 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.

4.1 Introduction

As clients continue to make use of dependency relationships to build software systems, software bots are increasingly being adopted to alleviate the cost of managing these dependencies. 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. 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 4.1 provides an example of a GKIR with the provider package name, current version, target version, and the dependency type in the client project highlighted.


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**.

Figure 4.1: 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.

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 (Jafari et al., 2020), 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 vulnerabilities, and more recent versions of providers often include fixes related to project stability (Cox et al., 2015). 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 found that bots are able to automate dependency updates (Mirhosseini and Parnin, 2017; Wessel et al., 2018), 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 chapter we perform an empirical study of four years of Greenkeeper data to examine the extent to which automated dependency management bots can either save or create unnecessary work in their client projects. 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 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 has 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[1].

The remainder of this chapter is organized as follows. Section 4.2 provides a detailed explanation on Greenkeeper. Section 4.3 explains the employed data collection procedures for the study. Section 4.4 presents the motivation, approach, and findings of our three research questions. Section 4.5 discusses the implications of our findings. Section 4.6 discusses the threats to the validity of our study. Finally, Section 4.7 concludes the chapter.

4.2 Greenkeeper

Greenkeeper is a popular dependency management bots that functions similarly to the description of automated dependency management bots provided in Section 2.2. In our study, we focus on data from Greenkeeper because the artifacts that 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

[1]https://github.com/SAILResearch/suppmaterial-22-ben-greenkeeper-overhead

opposed to Dependabot, for example, which is automatically enabled on client projects on GitHub to open security PRs[2], 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 dependency management bots mentioned in Section 2.2, as well as others not mentioned, 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 4.1.

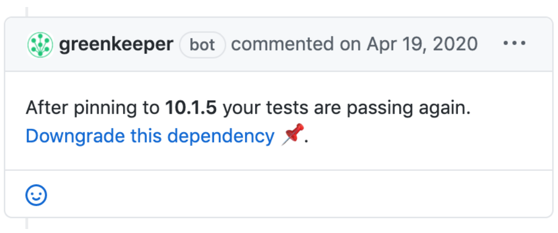
Since dependency updates that cause GKIRs to be created are *in-range breaking updates*, they can directly affect users of the client package if the offending dependency is a runtime dependency. Since the client's dependency constraints automatically allow for the new version of the dependency to be 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

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

client, and there is an incentive for the client developers to resolve them in a timely manner.

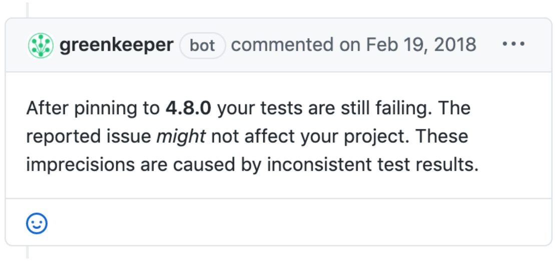
In addition to being alerted when a new GKIR is created in their project, clients will receive notifications for any activity that occurs on these GKIRs. One of the biggest drivers of these notifications is Greenkeeper itself. When a GKIR is first created, Greenkeeper will often attempt to pin the dependency 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 4.2 shows an example of Greenkeeper notifying the client that their tests are passing again (4.2a) versus that their tests are still failing (4.2b) with the affected provider version pinned to the previous release.

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 4.3 shows an example of Greenkeeper notifying the client that their tests are passing again (4.3a) versus that their tests are still failing (4.3b) with


greenkeeper bot commented on Apr 19, 2020

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.


greenkeeper bot commented on Feb 19, 2018

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.

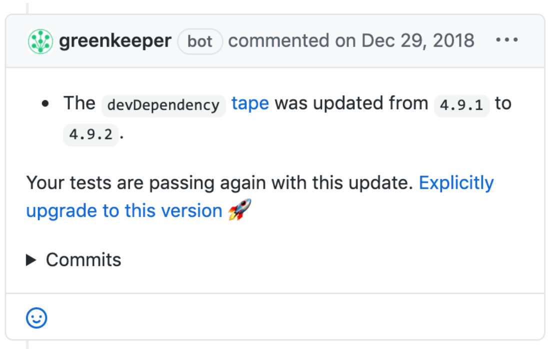
(b) A Greenkeeper comment indicating that the client's tests are still failing after pinning the provider to the previous version.

Figure 4.2: Examples of comments from Greenkeeper providing the results of automatically attempting to pin the provider when a GKIR is created.

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.

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*)[3]. For example, if a client were to depend on the Jest[4] 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

[3]https://greenkeeper.io/docs.html#monorepo-dependencies
[4]https://github.com/facebook/jest



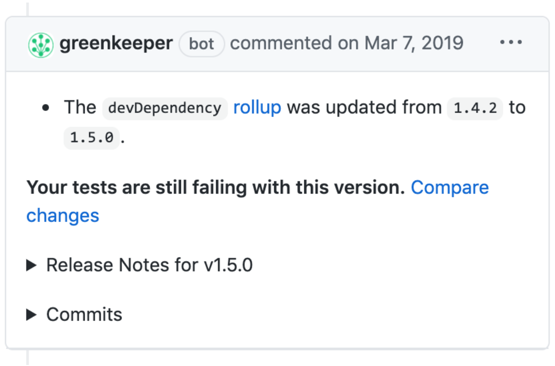
greenkeeper bot commented on Dec 29, 2018

- 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.



greenkeeper bot commented on Mar 7, 2019

- 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.

Figure 4.3: Examples of comments from Greenkeeper when a new version from a provider that caused a GKIR is released.

reduce the introduced overhead to the client, Greenkeeper will group releases from a predefined set of popular monorepo providers together (e.g., Angular[5], Babel[6], Jest, React[7], etc.) into bundled IRs.

4.3 Data Set

In this section, we discuss how we collect the data set to address the RQs that we outlined in the introduction. We use the workflow of Figure 4.4: (i) we identify projects on GitHub[8] that use the Greenkeeper bot, (ii) we collect all IRs for each identified project from the previous step, and extract the necessary information from each GKIR, (iii) we collect any supporting artifacts related to each identified

[5] https://github.com/angular/angular
[6] https://github.com/babel/babel
[7] https://github.com/facebook/react
[8] https://github.com/

GKIR from the previous step. Next, we provide a more in-depth explanation of each step in our data-collection workflow.

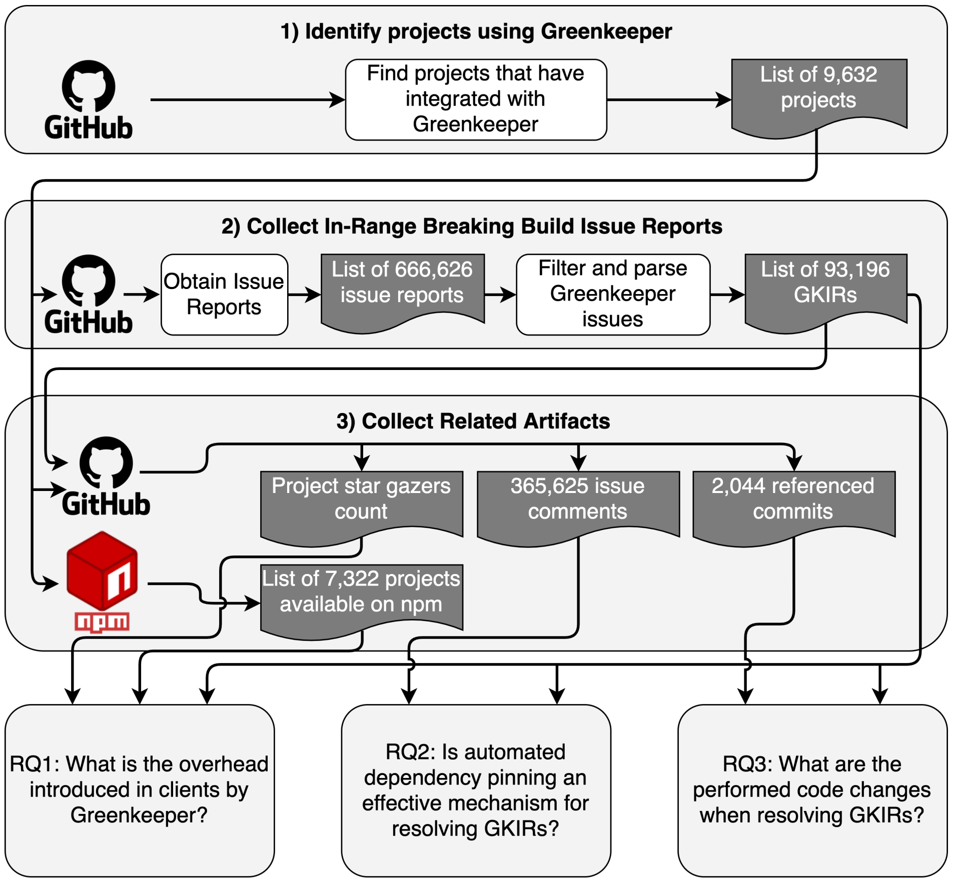

1) Identify projects using Greenkeeper
GitHub
Find projects that have integrated with Greenkeeper
List of 9,632 projects
2) Collect In-Range Breaking Build Issue Reports
GitHub
Obtain Issue Reports
List of 666,626 issue reports
Filter and parse Greenkeeper issues
List of 93,196 GKIRs
3) Collect Related 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?

Figure 4.4: Overview of the data collection process.

4.3.1 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 Search API[9] 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[10], 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.

4.3.2 Collect and parse GKIRs

To build our data set of IRs, we use the GitHub API [11] to retrieve all IRs opened in the list of projects we collected in Section 4.3.1. This step is necessary as a follow-up to the step in Section 4.3.1 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 4.3.1. GitHub considers PRs to be a type of issue, however we exclude PRs from our analysis, as our study focuses 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

[9] https://docs.github.com/en/rest/reference/search
[10] https://greenkeeper.io/docs.html#prerequisites
[11] https://docs.github.com/en/rest

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[12], 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[13].

[12]https://greenkeeper.io/docs.html#monorepo
[13]https://github.com/SAILResearch/suppmaterial-22-ben-greenkeeper-overhead

4.3.3 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 4.3.2. 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[14], 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 (Borges et al., 2016). 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

[14]https://docs.github.com/en/rest/reference/issues#events

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[15].

4.4 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.

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

Motivation. While software bots in general are useful for automating many tasks, prior research has shown that they have the potential side effect of disrupting developers in their work (Wessel and Steinmacher, 2020; Storey and Zagalsky, 2016). 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. (2018) 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

[15]https://github.com/SAILResearch/suppmaterial-22-ben-greenkeeper-overhead

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 4.4.1.1)?, 2) how long does it take for clients to address GKIRs and these related artifacts (Section 4.4.1.2)?, and 3) are GKIRs and these related artifacts actually useful to clients (Section 4.4.1.3)?

4.4.1.1 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 4.1)** ***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 4.5. This observation suggests that Greenkeeper is very prevalent in client projects that adopt it, and requires much attention from client developers.

Observation 4.2) ***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

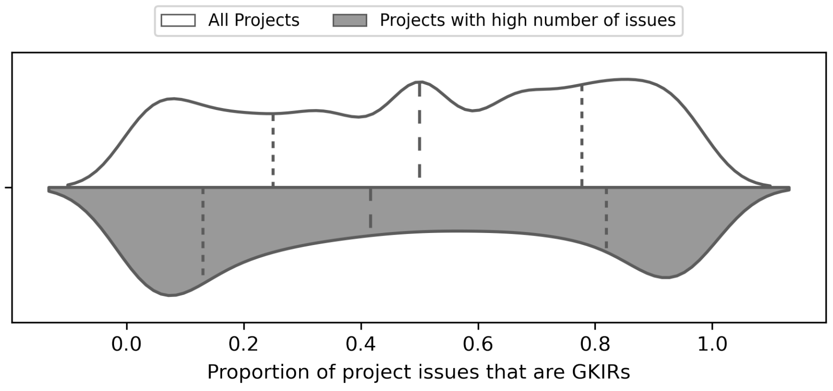


Figure 4.5: Violin-plot showing the distribution of the proportion of project issues that are in-range breaking build update issues. The dashed lines indicate the first quartile, median, and third-quartile.

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.

4.4.1.2 How long does it take for clients to address GKIRs and these related 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 (Borges et al., 2016), 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 ($\alpha = 0.05$) (Bauer, 1972) and correct the resulting p values using Bonferroni type adjustment (Armstrong, 2014). For statistically significant distributions, we assess the magnitude of the difference with the Cliff's Delta (d) estimator of effect size (Cliff, 1996). To classify the effect size, we use the following thresholds (Romano and Kromrey, 2006): 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 4.3)** ***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 that are 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 installations 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.

4.4.1.3 Are GKIRs and these related 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 4.1) 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 4.2).

Table 4.1: 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	*ˆAfter pinning to .* your tests are still failing*
Passing pin attempt	*ˆAfter 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 the 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 4.2. 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.

Findings. **Observation 4.4)** ***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 4.6 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,

Table 4.2: 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`

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 4.5) ***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

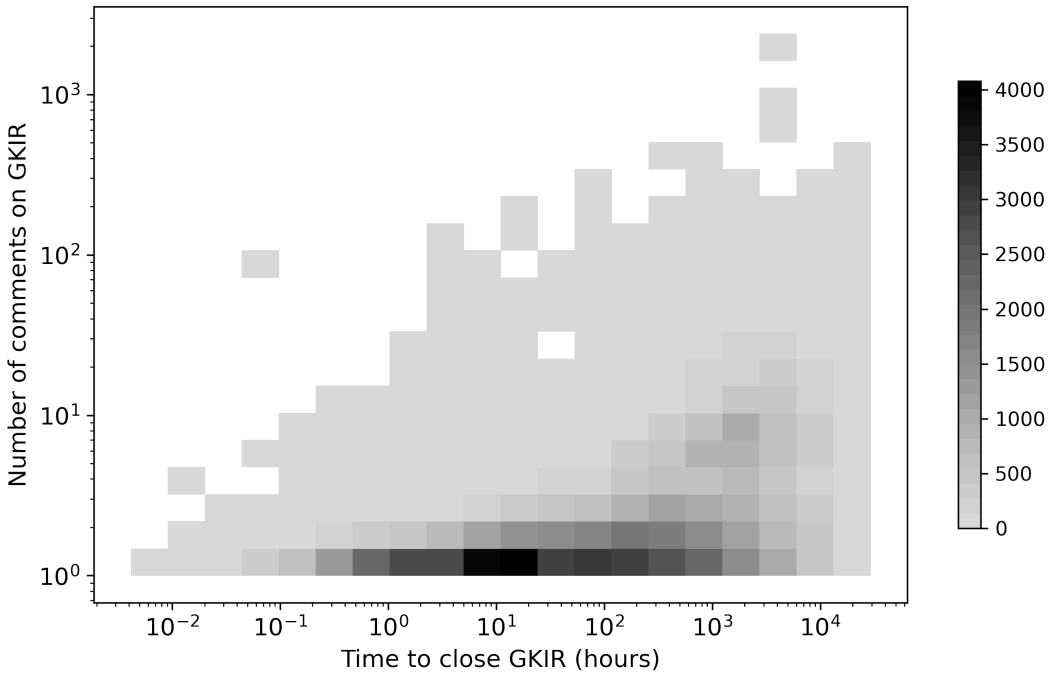


Figure 4.6: The distribution of the time taken to close a GKIR against the number of comments on the GKIR. The bin colour indicates the frequency of each data point.

same set of projects. We classify these comments (Figure 4.7) 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 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

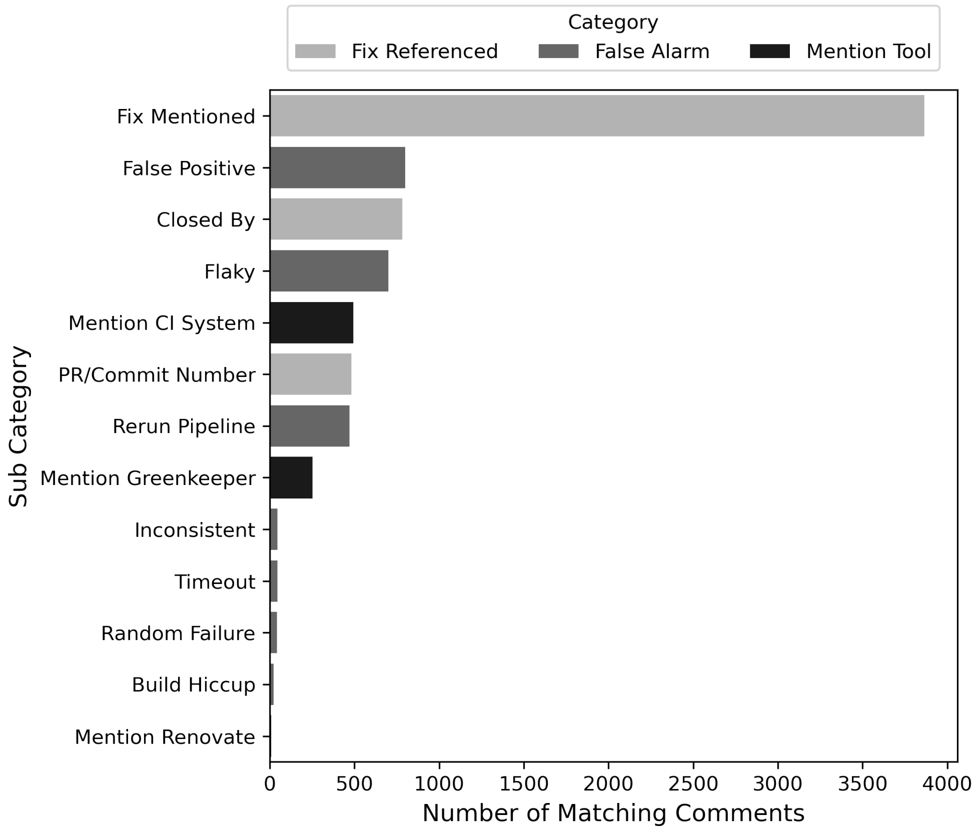


Figure 4.7: Bar-plot showing the number of comments matched to each of the patterns shown in Table 4.2.

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.

4.4.2 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 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 4.1.

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 $\pm 5\%$ 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 author of this thesis, both the author of this thesis and a collaborator of this study 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. Both the author of this thesis and the study collaborator 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 (Cohen, 1960). 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 (Fleiss and Cohen, 1973). In our case, the level of agreement is +0.93, which indicates that the classification results made by the author of this thesis are more likely to hold (Landis and Koch, 1977; Sim and Wright, 2005). Taking this high level of agreement into account, the author of this thesis then classified the remaining 85% of the random sample using the 10 categories agreed upon by both the author of this thesis and the study collaborator, which is a common process and has been done in previous work (DiStaso and Bortree, 2012; McDonald et al., 2019; Drouhard et al., 2017).

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 4.1 for alerting the client of a new dependency release.

Findings. **Observation 4.6)** ***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 4.3.

- **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 4.1 shows an example of a client's build[16] 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.

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

Table 4.3: 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.

Listing 4.1: 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[17] of Listing 4.2.

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

Listing 4.2: 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.
...
```

- **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[18] shown in Listing 4.3. Additionally, dependency versions can conflict, or sometimes cannot be found altogether, again causing the client's CI pipeline to fail.

[18] https://travis-ci.org/github/jaumard/trailpack-acl/jobs/529853292

Listing 4.3: 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 4.4 shows the build logs [19] where the client's CI pipeline failed due to a dependency not being initialized properly.

[19] https://travis-ci.org/github/chmanie/wdio-intercept-service/jobs/526469907

Listing 4.4: 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 4.5 shows an example excerpt of a client's build logs[20] where the build configuration file is missing completely, automatically causing the CI pipeline to fail.

[20] https://app.circleci.com/pipelines/github/unional/clibuilder/1/workflows/23256a75-d014-4d80-acf0-842644bfae24/jobs/1719

Listing 4.5: 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[21], causing the build to fail. Greenkeeper has since added native support for this feature[22]. However, failed CI pipelines caused by out-of-sync lockfiles remain common, an example of which is shown in the build logs[23] of Listing 4.6.

[21] https://blog.greenkeeper.io/greenkeeper-and-lockfiles-a-match-made-in-heaven-8260943fe521
[22] https://blog.greenkeeper.io/announcing-native-lockfile-support-85381a37a0d0
[23] https://travis-ci.org/github/travi/hapi-react-router/builds/619316527

Listing 4.6: 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 4.7 shows the build logs[24] of a client's test suite failing to run because of an error it the client's testing code.

[24]https://travis-ci.org/github/G5/gtm-controller/builds/463979922

Listing 4.7: 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 4.8 shows an example excerpt of a client's build logs[25] with this scenario. Additionally, network requests (e.g., download a dependency or upload code coverage statistics) can fail.

[25] https://travis-ci.org/github/visusnet/typereact/builds/549691105

Listing 4.8: 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 4.9 shows an example excerpt of a client's build logs[26] with this scenario.

Listing 4.9: 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.
...
```

[26]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 4.10 shows an example excerpt of a client's build logs[27] where the CI environment does not have the correct access rights to clone the project.

Listing 4.10: 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 4.7) ***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

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

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 project's 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.

4.4.3 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 (Jafari et al., 2020), as this type of versioning specification is often associated with outdated dependencies (Zerouali et al., 2018). However, 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 (Munson and Elbaum, 1998; Nagappan and Ball, 2005). 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[28] 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 4.8) ***The changes required to resolve GKIRs are similar to that of non-GKIRs.*** When comparing the number of file changes in commits

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

referenced by GKIRs and non-GKIRs, we find that they both change a median of 2 files per commit. Figure 4.8 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 ($|d| = 0.23$).

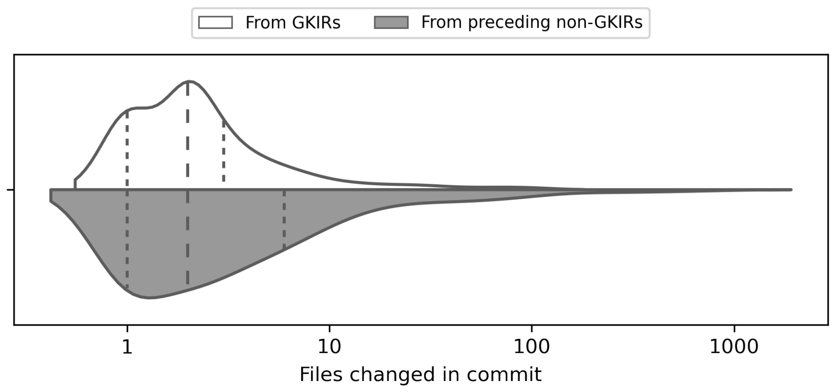


Figure 4.8: Violin-plot showing the distribution of number of files changed in commits referenced from GKIRs and commits referenced from non-GKIRs immediately preceding GKIRs. The dashed lines indicate the first quartile, median, and third-quartile.

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 4.9 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 4.9) ***More than half (56%) of changes that resolve GKIRs only include changes to dependency specification files.*** These changes are primarily

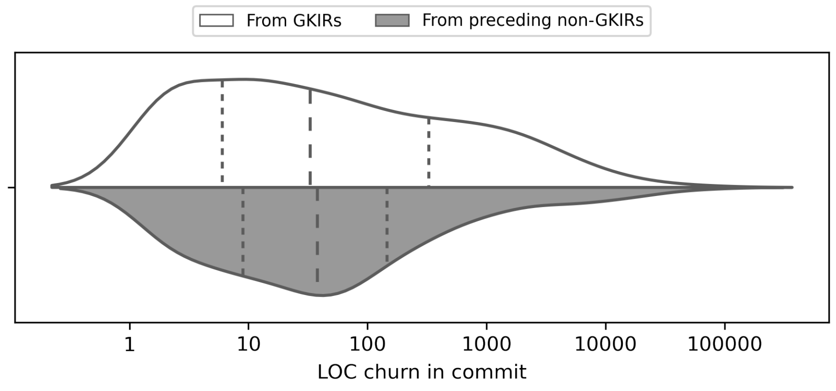


Figure 4.9: Violin-plot showing the distribution of lines of code (LOC) churn in commits referenced from GKIRs and commits referenced from non-GKIRs immediately preceding GKIRs. The dashed lines indicate the first quartile, median, and third-quartile.

made to the *package.json* file, which is manually maintained and is the 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 4.10 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`).

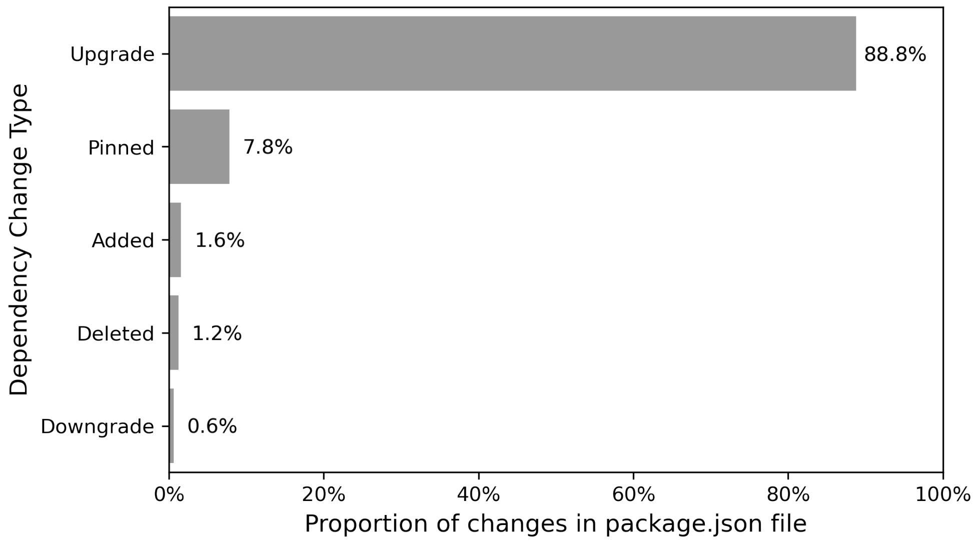


Figure 4.10: Bar-plot showing the proportion of the dependency change types made to the *package.json* file on commits referenced from GKIRs.

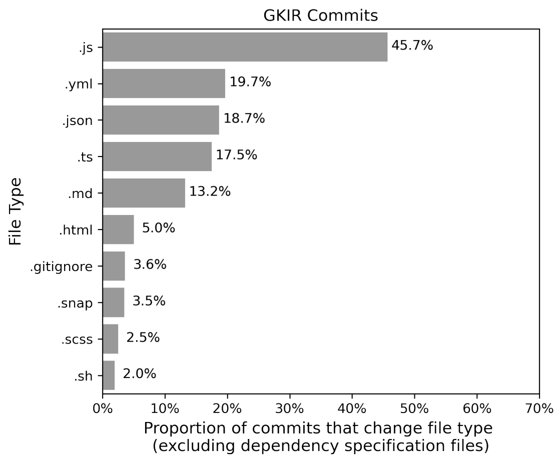
Additionally, we observe that clients may resolve multiple GKIRs in their project with a single patch. While the majority (79%) of commits only reference a single

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.

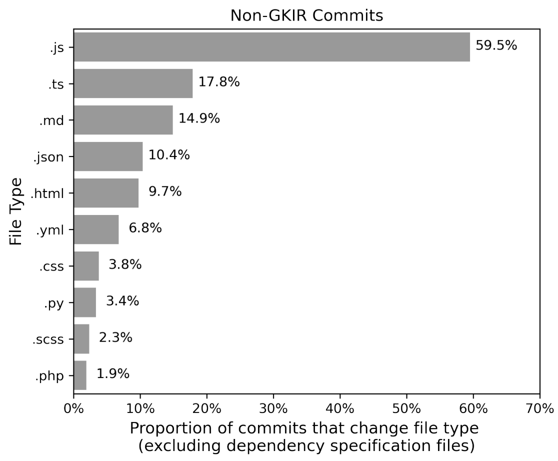
Observation 4.10) ***Commits referenced by GKIRs that do not only modify dependency specification files tend to include changes to a mixture of different file types, similar to commits referenced by non-GKIRs.*** While these commits commonly include changes to source code files (e.g., JavaScript and 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 4.11 shows the 10 most common file types that are changed in commits referenced by GKIRs (4.11a) and non-GKIRs (4.11b) 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.



(a) Commits referenced from GKIRs.



(b) Commits referenced from non-GKIRs.

Figure 4.11: 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*).

4.5 Discussion

In this section, we discuss the findings of Section 4.4. 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 4.5.1), as well as the current state-of-the-art in dependency management bots (Section 4.5.2).

4.5.1 Implications

Implication 4.1) ***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. [29], [30], [31], [32] 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.

[29] https://github.com/dependabot/dependabot-core/issues/2265
[30] https://github.com/dependabot/dependabot-core/issues/376
[31] https://github.com/dependabot/dependabot-core/issues/2526
[32] https://github.com/dependabot/dependabot-core/issues/1190

In fact, this issue has been a subject of discussion in at least four IRs [33], [34], [35], [36] 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

[33] https://github.com/dependabot/dependabot-core/issues/2265
[34] https://github.com/dependabot/dependabot-core/issues/376
[35] https://github.com/dependabot/dependabot-core/issues/2526
[36] https://github.com/dependabot/dependabot-core/issues/1190

multiple bots. [37], [38] 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 that are 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 in at least two IRs. [39], [40] 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 4.2) ***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

[37] https://github.com/dependabot/feedback/issues/954
[38] https://blog.mergify.io/replacing-dependabot-preview-auto-merge-feature/
[39] https://github.com/dependabot/dependabot-core/issues/2158
[40] https://github.com/dependabot/dependabot-core/issues/2189

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[41]. 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 4.3) ***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.

[41] https://dependabot.com/compatibility-score/

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 of causes of GKIRs with a failed pin attempt we reported in RQ2 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. 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 $\sim$`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 4.4) ***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[42] to determine the dependants of the client's package and monitor the publicly available data of build systems (e.g., TravisCI[43]) 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.

[42] https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph

[43] https://www.travis-ci.com/

Implication 4.5) ***Dependency management bots should take the type of CI failure into consideration when creating issues for new dependency releases.*** While we recommend in Implication 4.2 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 (2020) 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.

4.5.2 State-of-the-Art in Dependency Management Bots

While Greenkeeper was a very popular dependency management bot, creating over 130,000 PRs (Wyrich et al., 2021) and having been referenced in multiple studies (Mirhosseini and Parnin, 2017; Wyrich et al., 2021; Brown and Parnin, 2020), it has since been acquired by Snyk[44] and deactivated on June 3, 2020, and as such is no longer available for clients to integrate with on GitHub. However, Greenkeeper

[44]https://snyk.io/

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, Renovate, and Depfu[45]. 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 (Wyrich et al., 2021). 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 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

[45] https://depfu.com/

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 4.1. 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.[46]

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 4.1, 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 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`",

[46] https://github.com/dependabot/dependabot-core/issues/1190

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 behaviour by Greenkeeper aims to reduce the number of notifications received by clients[47], and encourages developers to use version range statements (rather than specific versions) for their dependency specifications[48]. Dependabot can be configured to act in a similar manner using its `versioning-strategy`[49] 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 that they would wish to ignore using the `suppressNotifications`[50] option. For example, clients can disable notifications from a PR being closed without being merged,

[47] https://github.com/greenkeeperio/greenkeeper/issues/990

[48] https://github.com/greenkeeperio/greenkeeper/issues/247

[49] https://docs.github.com/en/code-security/supply-chain-security/keeping-your-dependencies-updated-automatically/configuration-options-for-dependency-updates#versioning-strategy

[50] https://docs.renovatebot.com/configuration-options/#suppressnotifications

or can choose to not receive a warning notification for deprecated dependency releases. Renovate also includes a `stabilityDays`[51] 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 4.1), although it is unclear how often clients actually make use of them in practice.

Depfu has an update strategy called "reasonably up-to-date"[52] that clients can use to reduce the amount of generated activity 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[53], significantly reducing the amount of noise clients must deal with (Implication 4.1).

[51] https://docs.renovatebot.com/configuration-options/#stabilitydays
[52] https://depfu.com/blog/reasonably-up-to-date
[53] https://depfu.com/blog/reasonably-up-to-date

4.6 Threats to Validity

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

4.6.1 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 effect 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 threat 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 author of this thesis 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 author of this thesis and a collaborator of this study 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 (Cohen, 1960)), after which categories were consolidated as necessary. The level of agreement (+0.93) indicates that the classification results made by the author of this thesis are more likely to hold (Landis and Koch, 1977; Sim and Wright, 2005), and that the author of this thesis 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 (DiStaso and Bortree, 2012; McDonald et al., 2019; Drouhard et al., 2017).

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.

4.6.2 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 (Wyrich et al., 2021) and having been referenced in multiple studies (Mirhosseini and Parnin, 2017; Wyrich et al., 2021; Brown and Parnin, 2020), it has since been acquired by Snyk[54] 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

[54]https://snyk.io/

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[55] 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 management bots that followed were influenced by Greenkeeper. So, while our results may not generalize, 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 (Bogart et al., 2016). 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 study 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.

[55] https://depfu.com/

4.7 Chapter Summary

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 chapter, we describe 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.

CHAPTER 5

On bots leveraging the crowd for dependency management: An empirical study of the Dependabot Compatibility Score

This chapter is under review in the IEEE Transactions on Software Engineering (TSE).

SOFTWARE is increasingly being built by client packages making use of provider packages in the form of dependency relationships, which means client packages must face the essential and risky task of keeping their provider package dependencies up-to-date. Dependabot, a popular dependency management tool, includes a "compatibility score" feature that helps client packages assess the risk of accepting a dependency update by leveraging knowledge from "the crowd". For each dependency update, Dependabot calculates this compatibility score by dividing the number of successful updates by the total number of update attempts (candidate updates) made by other client packages that also use the same provider package as a dependency.

In this chapter, we describe our study on the efficacy of leveraging the crowd to help client packages assess the involved risks with accepting a dependency update. To accomplish this, we analyze 579,206 pull requests opened by Dependabot to update a dependency, along with 618,045 compatibility score records calculated by Dependabot. We find that the majority of compatibility scores do not have the minimum number of required candidate updates for the compatibility score badge to be shown on Dependabot pull requests. When the compatibility scores do have enough candidate updates, the vast majority of the scores are above 90%, suggesting that client packages should have additional angles to evaluate the risk of an update and the trustworthiness of the compatibility score. To overcome the lack of candidate updates when calculating a compatibility score, we propose metrics that amplify the input from the crowd and demonstrate the ability of those metrics to predict the acceptance of an update by client packages. We also verify that historical update metrics from client packages can be used to provide a more personalized compatibility score. Finally, we find that client packages should be hesitant to place total confidence in compatibility scores, as the candidate updates that are used to calculate the scores can be low both in quantity and quality. Based on our findings, we argue that, when leveraging the crowd, dependency management bots should (i) be mindful of ways to amplify the input from the crowd, (ii) consider historical metrics from the client package to provide a personalized compatibility score, (iii) include a confidence interval to help calibrate the trust clients should place in the compatibility score, and (iv) take into consideration the quality of tests that exercise candidate updates to avoid biasing the compatibility score.

5.1 Introduction

As software is increasingly being built by making use of dependency relationships, an important development decision that is faced by clients is whether to update the provider package from the presently used version in their package (i.e., the *origin version*) to the newest release of the provider (i.e., the *target version*). Doing so allows clients to receive the aforementioned potential benefits, but at the risk of these new versions modifying existing functionality or introducing API-backwards incompatibilities (a.k.a., breaking updates) (Bogart et al., 2016). One strategy employed by client packages to protect against breaking updates is to run their own continuous integration (CI) pipeline, including unit and integration tests, against newly released versions of their dependencies (Hilton et al., 2016). Unless an update is intentionally breaking backwards compatibility (e.g., a major release), the client's CI pipeline should continue to pass with the new release applied (Raemaekers et al., 2017).

However, many client packages do not have a full CI pipeline enabled (Hilton et al., 2016), and therefore are unable to automatically test whether a dependency update will be compatible with their package. One strategy that attempts to address this issue is to leverage knowledge from the crowd to provide insights about the risk of a newly released version of a provider package. In fact, Mujahid et al. (2020) and Mezzetti et al. (2018) both propose techniques that leverage the test suites of clients of a provider package in an effort to detect breaking changes in new releases of the provider package, and use these test outcomes as crowd-sourced indicators of the risk of adopting said provider release.

Dependabot is an automated dependency management tool that packages on GitHub can integrate with to automate the process of updating and testing new releases from provider packages. Dependabot sits between a client's package manager and GitHub, observing all the provider packages the client package depends on. Each time one of these providers release a new version, Dependabot opens a pull request (PR) in the client package with the dependency update, and the client's CI pipeline is run automatically on the updated dependency to test if it is a breaking update. Dependabot records the result of updating the dependency, and calculates a *compatibility score* for the provider package release as the percentage of PRs with a successful CI conclusion (*successful updates*) to the total number of PRs updating between the origin and target versions of said provider (*candidate updates*). This compatibility score is shown as a badge on PRs that are opened by Dependabot for the same provider update, and is meant to give practitioners a sense of the involved risk when updating a dependency by leveraging the knowledge of "the crowd", so that clients can be confident a new provider version is backwards compatible and bug-free.

However, this technique has its limitations, as it requires a crowd of a large scale to work effectively, and there is a lack of research that examines the value and challenges of using this approach. On the one hand, since Dependabot is state-of-the-art and the most widely used automated dependency management bot leveraging the idea of crowd-based risk assessment, clients stand to reap the benefits of these indicator metrics to help them keep their dependencies up-to-date and their packages in working order. On the other hand, it is unknown whether the crowd is actually able to provide a strong enough signal for Dependabot to be able to calculate trustworthy compatibility scores, nor the level of confidence clients should actually place in these compatibility scores.

Therefore, in this chapter, we describe our study that explores the efficacy of Dependabot's strategy of leveraging the crowd to provide a compatibility score to help clients assess the involved risks with dependency management. In the following, we list our research questions and key observations:

RQ1: Does the crowd provide enough support to calculate a trustworthy compatibility score? We examine the proportion of compatibility scores that have the minimum number of candidate updates required by Dependabot and the range of scores practitioners most often see when they receive a Dependabot PR. Our results indicate that compatibility scores tend to have a small number of candidate updates and are heavily skewed towards 100%. Therefore, clients should be hesitant to trust compatibility scores, and other sources of information should be considered to calculate compatibility scores to overcome the lack of candidate updates.

RQ2: Which other sources of information can be considered when the crowd does not provide enough support to calculate a compatibility score? We examine seven features across two dimensions: i) *origin version range compatibility scores*, which considers the candidate updates from a range of origin versions of a provider package (e.g., `2.0.x`) that have been updated to a specific target version (e.g., `2.0.4`) and aims to amplify the knowledge from the crowd, and ii) *client history of updates*, which aim to capture the historical stability of the client's package in general, the historical compatibility of the provider package with the client's package, and the historical level of confidence the client package places in the provider package. We observe that considering a range of origin versions can increase the number of candidate updates that are used to calculate compatibility scores. We also find that features from both of the aforementioned dimensions can result in models that predict whether a dependency update will be accepted or rejected by a client package with an AUC of 0.64-0.80, with historical metrics from the client package tending to have the highest predictive power.

RQ3: How much confidence should client packages place in the compatibility score? We evaluate the confidence Dependabot has in compatibility scores by building an associated confidence interval for each compatibility score. We observe that half of compatibility scores with at least 5 candidate updates have a confidence interval whose bounds are further than 15% from the compatibility score. We also explore the quality of checks that make up the CI pipelines of candidate updates, and find that candidate updates that contribute to compatibility scores may not always truly test the associated dependency update.

The aforementioned results led us to conclude that, while popular dependency management bots like Dependabot making use of the crowd to assess the compatibility of a dependency update is a promising strategy, the compatibility scores are often not available, and, even when the scores are available, can be misleading for clients without the support of a confidence interval. Additionally, bots should employ further methods to help amplify the input from the crowd or consider historical upgrade metrics to assess whether a client package should accept or reject a dependency update.

More generally, the main contributions of this study are: (i) an empirical study that examines Dependabot's current strategy of leveraging the crowd to provide a compatibility score to help clients assess the involved risks with accepting a dependency update, (ii) a description and evaluation of additional data sources that can be considered when the crowd does not provide enough support to calculate a compatibility score, (iii) a description and evaluation of an approach to help calibrate the level of trust clients should place in the score, (iv) a series of practical recommendations for designers of automated dependency management bots on effectively leveraging the crowd to help clients assess the risk of accepting a dependency update, and (v) a supplementary material package with the data that is used in this study[1] as a means to bootstrap other studies in the area.

The remainder of this chapter is organized as follows. Section 5.2 introduces key concepts related to our study. Section 5.3 explains the employed data collection procedures. Section 5.4 presents the motivation, approach, and findings of our three research questions. Section 5.5 discusses the implications of our findings. Section 5.6 discusses the threats to the validity of our study. Finally, Section 5.7 concludes the chapter.

5.2 Background and Motivating Example

In this section, we present the key concepts related to automated dependency management with Dependabot (Section 5.2.1), as well as existing studies in the field of

[1] https://github.com/SAILResearch/suppmaterial-22-ben-dependabot_compatibility_score

crowd-sourced software engineering (Section 5.2.2). We also present a motivating example (Section 5.2.3) to help illustrate the intentions of the compatibility score.

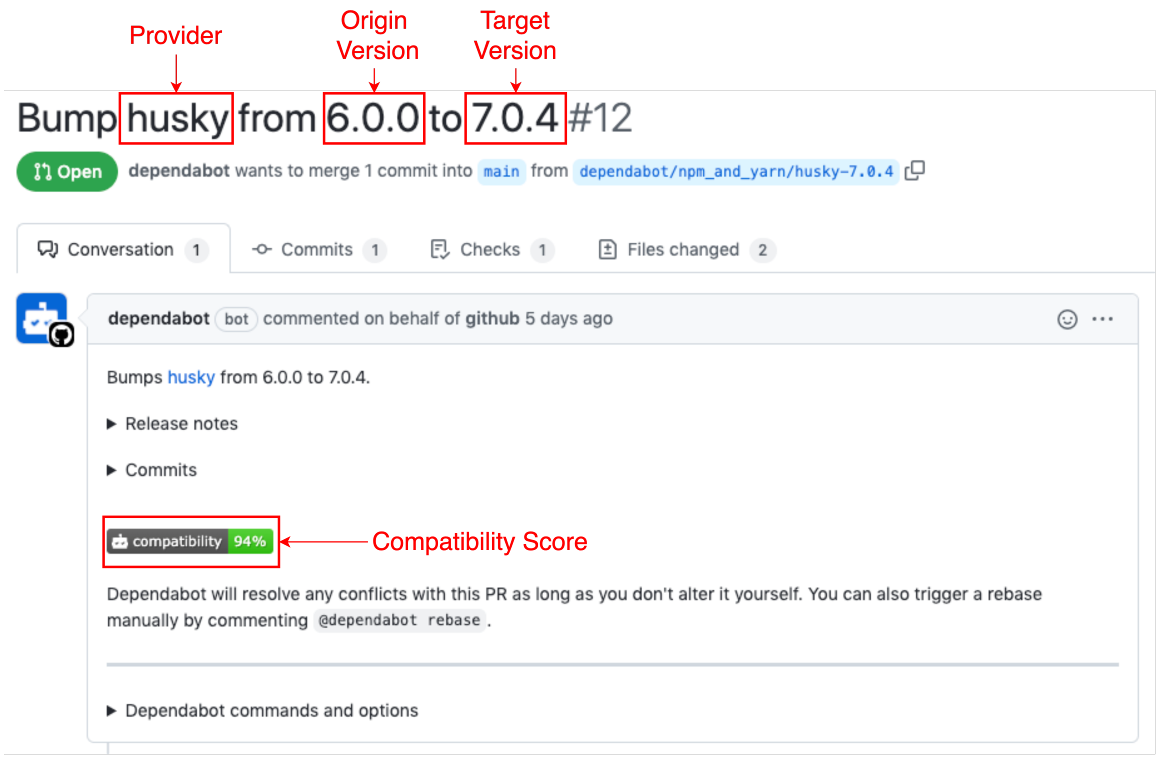
5.2.1 Dependabot

Dependabot is perhaps currently the most popular automated dependency-management tool, having first launched on May 26, 2017[2] and later being acquired by GitHub on May 23, 2019[3]. Dependabot supports a wide range of different language ecosystems, including JavaScript, Ruby, and Python to name a few. Dependabot sits between a package manager and GitHub, observing all the providers on which a client package depends. Each time one of these providers release a new version, Dependabot opens a new PR with the client's dependency specifications updated to accept the newly released provider version. Once a Dependabot PR is created, the client's CI pipeline, if configured, runs automatically against the PR branch to determine if the new version of the provider passes all the client's tests. The client can then decide whether they would like to accept or reject the Dependabot PR.

To support clients in their decision of whether they should accept or reject a Dependabot PR, Dependabot includes a summary statistic called a *compatibility score* that leverages knowledge from the crowd to provide insights about the risk of a newly released version of a provider package. When a new provider version is released, Dependabot creates similar PRs across multiple client packages to update the provider from the origin version used by each client to the target version which has been newly released by the provider. More formally, the provider package named P, origin version V_O, and target version V_T create a 3-tuple for the

[2]https://dependabot.com/blog/introducing-dependabot/
[3]https://dependabot.com/blog/hello-github/

dependency update (P, V_O, V_T). For each client with CI enabled (e.g., Travis CI[4] or GitHub Actions[5]) and a previously passing test suite, Dependabot records whether the 3-tuple dependency update breaks any of the client's tests. Dependabot considers PRs that meet this criteria to be *candidate updates*. Dependabot considers a candidate update to be a *successful update* if the client's CI pipeline is in a passing state with the dependency update. Figure 5.1 provides an example of a Dependabot PR with the provider package name, origin version, target version, and the compatibility score highlighted.



Bump husky from 6.0.0 to 7.0.4 #12

dependabot wants to merge 1 commit into `main` from `dependabot/npm_and_yarn/husky-7.0.4`

dependabot commented on behalf of github 5 days ago

Bumps husky from 6.0.0 to 7.0.4.

▶ Release notes

▶ Commits

Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting `@dependabot rebase`.

▶ Dependabot commands and options

Figure 5.1: An example of a Dependabot pull request with the provider package name, origin version, target version, and the compatibility score highlighted.

[4]https://travis-ci.com/
[5]https://docs.github.com/en/actions

The compatibility score of a dependency update is the percentage of CI runs that passed when updating the dependency between the same origin and target versions (i.e., the number of successful updates divided by the number of candidate updates for the origin version and target version of the provider package). It is important to note that the client does not necessarily have to merge the Dependabot PR in order for it to be considered a candidate update and contribute to the compatibility score. In order for the compatibility score to correctly show up on the Dependabot PR, the dependency update must have at least 5 candidate updates[6]. Otherwise, the badge will simply say that the compatibility score is "unknown".

5.2.2 Crowd-sourced Software Engineering

Several researchers have studied how to use the "wisdom of the crowds" to help in the software engineering domain. Stack Overflow[7], a popular Question and Answer (Q&A) site, has been the topic of many studies (LaToza and van der Hoek, 2016; Treude et al., 2011; Rosen and Shihab, 2016; Barua et al., 2014; Vasilescu et al., 2013) that explore how the job of answering questions related to software engineering is outsourced to the crowd. Specifically, Abdalkareem et al. (2017b) analyze 1,414 Stack Overflow related commits and observe that developers use this crowd based knowledge mostly for technical comprehension, collecting users' feedback and code reuse.

The idea of using the crowd to help with dependency management has also been studied. Mileva et al. (2009) proposed an approach and associated tool to

[6] https://github.com/dependabot/dependabot-core/issues/4001#issuecomment-870399478

[7] https://stackoverflow.com/

help client packages decide when to use which version of a provider package. They reason that if a provider package version is used by more client packages, it should be more likely to be recommended. To use the crowd to detect breakage changes in new provider releases, Mujahid et al. (2020) propose a technique that leverages the automated test suites of other client packages that make use of the same dependency to test newly released versions, describing essentially an academic version of the Dependabot compatibility score. They find that this crowd-based approach can detect six of ten breakage-inducing versions they studied, and that their findings can help clients to make more informed decisions when they update their dependencies, which is also the goal of the Dependabot compatibility score. Similarly, Mezzetti et al. (2018) describe an approach called *type regression testing* to automatically detect type-related breaking changes. This approach leverages the tests of clients of a provider package to construct models of new releases of provider package APIs, and then compare these models to detect potential *type regressions*, demonstrating that this approach can detect type-related breaking changes with high accuracy. They argue that using the clients' test suite, rather than the test suite of the provider package itself, is more likely to provide representative executions and only use the public parts of the provider package.

5.2.3 Motivating Example

To help illustrate how the compatibility score is used in practice and how it can be misleading in the context of dependency management, we provide a simple motivating example.

Alice is a software developer responsible for developing and maintaining an application in her company. In order to enable code reuse and speed up development time, she relies on a few third-party packages to accomplish specific tasks in her application. However, in order to keep up with the demanding timeline of her employer, she has only managed to build a CI testing pipeline that amounts to "smoke tests"[8] (i.e., a non-exhaustive set of tests that aim at ensuring only the most important functions of her application work). She has not written any tests that exercise the portions of her application that make use of her dependencies, as she figures that these packages would be "deployment tested" (i.e., they are extensively used in production by many clients).

As with many provider packages, the dependencies Alice uses get updated frequently. Alice wants to be more proactive in managing her software dependencies, so she uses Dependabot to automatically open PRs to update her dependencies as new releases become available. She knows that Dependabot is the most commonly used automated dependency management bot and enjoys the convenience of being notified when her dependencies become out-of-date.

While Alice very much wants to keep her dependencies up-to-date, she is aware of the involved risks with blindly accepting a new update. She has heard stories from other developers who have had to drop all of their work in order to fix a broken CI pipeline caused by a dependency update. Even worse, Alice has even heard of developers who were only informed by their customers that their application was broken weeks after deploying a new version that contained a breaking dependency update. She could only imagine the amount of work that it took to find that this particular dependency update was the root cause, not to mention the user's perceived

[8] https://softwaretestingfundamentals.com/smoke-testing/

lack of quality that comes with deploying a broken version of the application. Since Alice knows her test suite does not sufficiently cover her application, she thinks the compatibility score badge Dependabot includes on the PRs is a very helpful indicator for the compatibility of the dependency update, and tends to rely on it when deciding whether to accept a dependency update.

One day, Alice sees that Dependabot has opened a PR in her application. After briefly examining the PR, she sees that her CI tests pass when applied against the dependency update, but that Dependabot has not been able to calculate a compatibility score for the update. Knowing that her tests are most likely not capable of exercising major portions of her dependencies, she decides to hold off on taking any action on this PR.

After a few days, Alice checks back on the PR, and finds that Dependabot reports that updating the dependency from the version that she currently uses to the newly released target version has a compatibility score of 100%. With this information in mind, she decides to merge the PR.

A few days later, Alice gets a message from her boss stating that their application is experiencing some unexpected behaviour. After debugging, Alice finds that the recent change she made by merging the Dependabot PR for the dependency update introduced the issue. Even though her CI testing pipeline passed, the tests were not able to detect the breaking behaviour in the updated dependency - the tests simply did not cover the case causing the unexpected behaviour. Remembering the 100% compatibility score she saw when she merged the PR, she investigates and discovers that the dependency update only had 5 candidate updates - one of which was actually the PR Dependabot opened for her own application! Alice's confidence in the compatibility score has been severely shaken after this incident. She made the wrong decision because she didn't know "how much" she could trust the compatibility score when she merged the Dependabot PR, and now no longer believes the compatibility score to be a reliable metric.

5.3 Data Collection

In this section, we discuss how we collect the dataset to address the RQs outlined in the introduction. We use the workflow of Figure 5.2: (i) we identify packages on GitHub that use Dependabot, (ii) we collect all Dependabot PRs for each package that is identified in the previous step, and extract the necessary information and collect related artifacts for each Dependabot PR, (iii) we collect the compatibility scores for the provider package updates that are related to each Dependabot PR identified in the previous step, (iv) we build two distinct datasets using the data collected in the two previous steps.

Next, we provide a more in-depth explanation of each step in our data-collection workflow.

5.3.1 Identify packages using Dependabot

To identify the packages using Dependabot, we leverage the Google BigQuery Public Datasets[9] to search for commits on GitHub that have been authored by Dependabot. Each of these commit records contains the parent package name on GitHub, which we use to build our list of packages to include in our study. It is known

[9]https://cloud.google.com/bigquery/public-data/

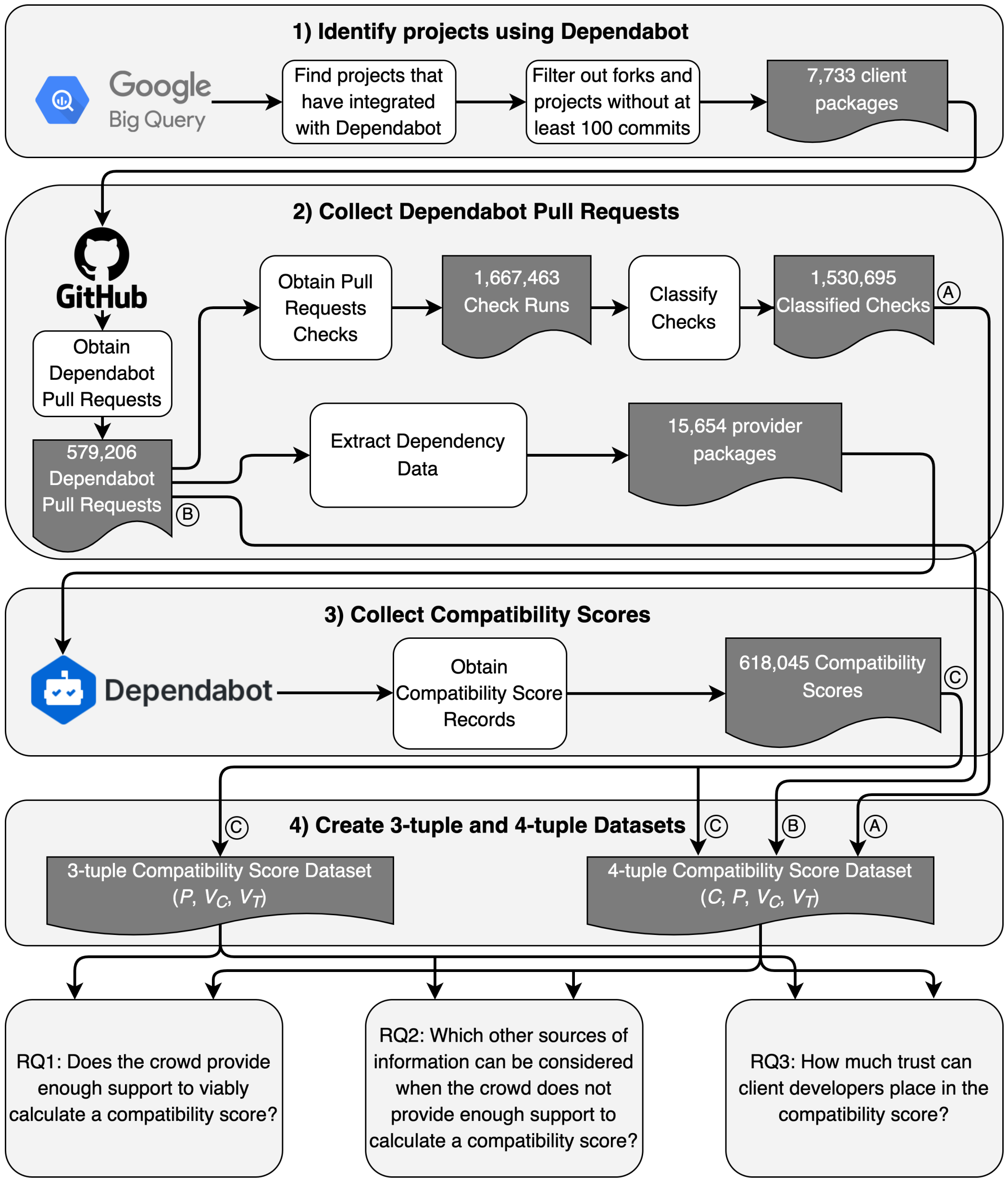

1) Identify projects using Dependabot
Google Big Query
Find projects that have integrated with Dependabot
Filter out forks and projects without at least 100 commits
7,733 client packages
2) Collect Dependabot Pull Requests
GitHub
Obtain Dependabot Pull Requests
579,206 Dependabot Pull Requests
Obtain Pull Requests Checks
1,667,463 Check Runs
Classify Checks
1,530,695 Classified Checks
Extract Dependency Data
15,654 provider packages
3) Collect Compatibility Scores
Dependabot
Obtain Compatibility Score Records
618,045 Compatibility Scores
4) Create 3-tuple and 4-tuple Datasets
3-tuple Compatibility Score Dataset (P, V_C, V_T)
4-tuple Compatibility Score Dataset (C, P, V_C, V_T)
RQ1: Does the crowd provide enough support to viably calculate a compatibility score?
RQ2: Which other sources of information can be considered when the crowd does not provide enough support to calculate a compatibility score?
RQ3: How much trust can client developers place in the compatibility score?

Figure 5.2: Overview of the data collection process.

that GitHub contains some toy packages (Kalliamvakou et al., 2014) which are not representative of the software packages we aim to investigate. Therefore, once the dataset of packages using Dependabot is collected, we apply some filtering criteria for selecting a set of packages with a history of activity. We only include packages that are non-forked and contain at least 100 commits, as recommended by prior studies (Alfadel et al., 2021; Mirhosseini and Parnin, 2017; Kalliamvakou et al., 2014). In total, we extract a list of 7,733 GitHub packages that meet our filtering criteria. Due to our filtering criteria, this is by no means an exhaustive list of all packages that use Dependabot.

5.3.2 Collect Dependabot Pull Requests

We use the GitHub API[10] to retrieve all Dependabot PRs opened in the list of packages that we collected in Section 5.3.1. This step is necessary as a follow-up to Section 5.3.1 as it allow us to extract the information about the provider Dependabot is attempting to update with the PR. Overall, we collect a total of 579,206 PRs opened by Dependabot for the time period between June 2017 and June 2021.

Dependabot includes information on the updated provider package in the title of the PR (see Figure 5.1). We extract the provider package name, the origin version of the provider used by the client, and the newly released target version of the provider from the PR title using a set of regular expressions, which we use to collect compatibility scores in Section 5.3.3. We are able to extract this information from 575,860 (99.4%) of the PRs. Upon closer examination of the PRs for which we are not able to extract this information for, we find that these PRs are not in fact dependency updates, but rather automatic PRs created by Dependabot to create or modify the Dependabot configuration file in the client package, or to update Dependabot itself.

[10] https://docs.github.com/en/rest

To determine whether a client's CI pipeline passed or failed when testing each dependency update, we retrieve the GitHub Checks[11] that ran on each PR. Checks are pipeline runs or custom scripts that perform specific tasks (e.g., linters[12] or Travis CI builds), and they are used by Dependabot to determine the result of a candidate update (i.e., success or failure). There can be multiple checks that run against a PR. For example, a client might have a check that ensures the client's package builds, another check to run the test suite, and a final check to detect and fix any code style issues. The set of checks that run against a PR make up the CI pipeline for the client's package. Overall, we find that 38% of Dependabot PRs (or 43% of client packages) have a configured CI pipeline (i.e., a set of checks) to run on new PRs, which is in line with prior results by Hilton et al. (2016) when examining the percentage of open-source packages that have a CI pipeline configured.

Hejderup and Gousios (2021) find in their study that it is common for clients to have a low quality set of tests that run against dependency updates. With these findings in mind, we decide to classify the types of checks to determine what types of CI pipelines are run against Dependabot PRs. We use the name of the check, which is used to give a high-level description of the task the check performs, to assign each check to a specific overarching category. We match 91.8% of the checks using the process described in Appendix A to one of the following six categories: *Build* (58.1%), *Test* (17.2%), *Useless* (11.2%), *Lint* (7.0%), *Deploy* (4.9%), and *Security Analysis* (1.6%). We find that clients typically group their entire build, test, and deploy pipeline into a single check workflow, which explains why the *Build* category is the most common. The *Useless* category consists of checks that do not help

[11] https://docs.github.com/en/rest/reference/checks
[12] https://github.com/collections/clean-code-linters

with determining whether the changes contained in the PR are compatible with the client package (e.g., automatically adding a label to the PR or uploading build logs to a separate repository).

5.3.3 Collect Compatibility Scores

For each specific provider package we extracted in Section 5.3.2, we retrieve all compatibility scores using the Dependabot API[13]. The Dependabot API requires a package manager and a provider package name as query parameters, and returns the compatibility score records for all 3-tuple update combinations that Dependabot has recorded for that provider package. Each compatibility score record contains the number of candidate updates and the number of successful updates Dependabot has recorded for the 3-tuple dependency update in question. Overall, we collect the compatibility score records for a total of 618,045 3-tuple dependency updates.

5.3.4 Create 3-tuple and 4-tuple Datasets

Because our package list in Section 5.3.1 is a non-exhaustive set of package that use Dependabot, the Dependabot PRs we collect in Section 5.3.2 do not represent the full list of the number of candidate updates that are used by Dependabot to calculate the compatibility scores. Therefore, the process described in Section 5.3.3 is necessary to get the complete picture in terms of the number of candidate and successful updates contributing to the associated compatibility score for each 3-tuple update. Hence, we refer to the compatibility scores we collect in Section 5.3.3 as the "3-tuple dataset".

[13]https://dependabot.com/compatibility-score/

However, records from the 3-tuple dataset only contain the compatibility scores for the 3-tuple update in question, without any relating information on the specific Dependabot PRs opened in client packages that are contributing as candidate or successful updates to these scores. As a result, we are not able to study the relationship between compatibility scores and the associated merge status of candidate update Dependabot PRs using the 3-tuple dataset.

Therefore, we link the compatibility score from the 3-tuple dataset with the associated Dependabot PRs (if present) we collected in Section 5.3.2. This allows us to link specific candidate update Dependabot PRs to a compatibility score and provides a means to study the relationship between the compatibility scores and the merge status of said Dependabot PRs. With the specific client linked to the 3-tuple dependency update for the compatibility score, we form a 4-tuple consisting of (C, P, V_O, V_T), where C is the client package from a Dependabot PR, P is a provider package used by C, V_O is the origin version of P used by C at the time the Dependabot PR was opened, and V_T is the newly released target version of P at the time the Dependabot PR was opened (notice that P, V_O, and V_T form a single record from the 3-tuple dataset). Hence, we refer to this dataset as the “4-tuple dataset”.

An additional description on how and why the 3-tuple and 4-tuple datasets may differ is included in Appendix B.

5.4 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.

5.4.1 RQ1: Does the crowd provide enough support to calculate a trustworthy compatibility score?

Motivation. Client packages often want to be aware of the risk of a dependency update breaking the build (Bogart et al., 2015), particularly when the quality of test suites cannot be fully trusted – a relatively common scenario according to recent research (Hejderup and Gousios, 2021). Dependency bots (e.g., Dependabot) have recently integrated a new feature that leverages crowd-sourced information to estimate the risk of an update in the form of a compatibility score. However, the viability of the compatibility score has yet to be studied in practice, and it is unclear whether Dependabot does in fact create enough dependency updates to be able to effectively determine a consensus from the crowd about whether a dependency update is safe or not, as well as whether client packages can rely on this consensus. In fact, these issues have been the source of complaints on the Dependabot repository[14] as well as developer blog sites[15]. Therefore, in this research question, we look to answer i) how often do compatibility scores have the minimum number of candidate updates required to be shown as a badge on Dependabot PRs? and ii) when the badge is shown on Dependabot PRs, is the distribution of scores seen by client packages useful to assess the risk of an update?

Approach. To determine how often a known compatibility score shows up on Dependabot PRs, we examine the proportion of candidate updates each dependency update has. Recall that in order for a known compatibility score badge to show up

[14]https://github.com/dependabot/dependabot-core/issues/2443
[15]https://dev.to/lhuria94/comment/ofe5

on the PR, the dependency update must have at least 5 candidate updates. Otherwise, the badge will simply say that the compatibility score is “unknown”. We then examine the distribution of compatibility scores with at least 5 candidate updates to determine the range of scores practitioners most often see when they receive a Dependabot PR. We perform this analysis on the compatibility score for both the 3-tuple and 4-tuple datasets.

Findings. **Observation 5.1)** ***The majority (83%) of dependency updates do not have enough candidate updates to display a compatibility score badge on Dependabot PRs.*** Figure 5.3 shows the distributions of candidate updates for compatibility scores with at least 1 candidate update from both the 3-tuple and 4-tuple datasets. When examining the distribution of candidate updates for compatibility scores from the 3-tuple dataset, we find that only 17% have at least 5 candidate updates. This finding was surprising, as it shows that, even though Dependabot may be opening hundreds of PRs when a provider package releases a new version, more than four-fifths of the associated compatibility scores are still not shown on these PRs simply because the “consensus from the crowd” doesn’t exist for the dependency update. In reality, this proportion is likely lower, as all compatibility score records in our 3-tuple dataset must have at least 1 candidate update (see Section 5.3.3), which means we do not include the compatibility scores for dependency updates that have no candidate updates in our analysis.

We find that 43% of the compatibility scores from the 4-tuple dataset do not have enough candidate updates for the compatibility score to show up on Dependabot PRs, with the median number of candidate updates being 41. Recall that compatibility scores from the 4-tuple dataset include compatibility scores for dependency updates from provider packages used by active clients, as well as commonly used origin and target versions of these provider packages. Therefore, we expect these compatibility scores to have a higher number of candidate updates than those from the 3-tuple dataset. Although we observe this improvement compared to the 3-tuple dataset, it must be considered that, while these may be popular dependency updates, only 57% have a compatibility score with enough candidate updates to be shown on the associated Dependabot PR. Our observations suggest that alternative sources of information should be considered to support dependency updates when the crowd does not provide enough input to calculate a compatibility score.

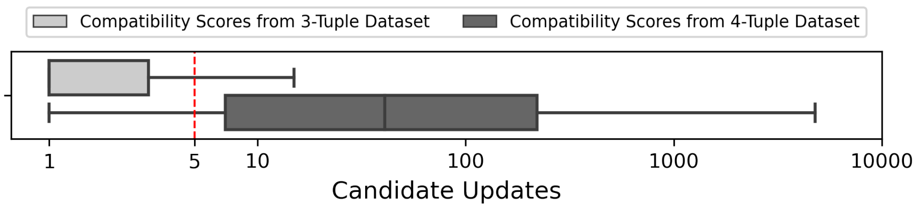


Figure 5.3: The distribution of candidate updates for compatibility scores with at least 1 candidate update from the 3-tuple and 4-tuple datasets.

Observation 5.2) ***Client packages are usually forced to distinguish between only a small range of compatibility scores.*** When the badge with the compatibility score is shown on the Dependabot PR (i.e., the compatibility score has at least 5 candidate updates), we find that the vast majority of compatibility scores (76% and 89% in the 3-tuple and 4-tuple datasets, respectively) are greater than 90%.

Figure 5.4 shows the distributions of compatibility scores that have at least 5 candidate updates from both the 3-tuple and 4-tuple datasets. We can see that, with so many compatibility scores grouped at the high end of the score range, it can be difficult for client packages to distinguish between such a small range of scores, and in fact may be misled into thinking dependency updates are more compatible than they actually are. In order to help calibrate clients' trust on the usually excessively high compatibility scores, additional supporting metrics, such as the adoption of an accompanying confidence score for each compatibility score, might be useful.

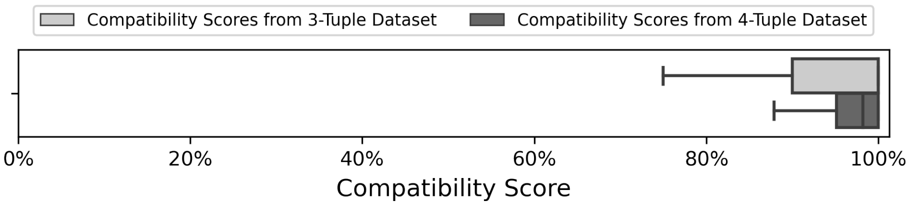


Figure 5.4: The distribution of compatibility scores that have at least 5 candidate updates from the 3-tuple and 4-tuple datasets.

RQ1: Does the crowd provide enough support to calculate a trustworthy compatibility score?

- The majority of compatibility scores do not have the minimum number of candidate updates to be shown correctly on Dependabot PRs.
- The vast majority of the shown scores are above 90%, hindering clients' ability to differentiate the risks of a dependency update.

5.4.2 RQ2: Which other sources of information can be considered when the crowd does not provide enough support to calculate a compatibility score?

Motivation. We found in RQ1 that the majority of dependency updates do not have enough candidate updates recorded by Dependabot to correctly show a compatibility score badge on PRs. Dependabot is the most popular dependency management bot available in the open-source community, and it is unlikely that there is another tool that will be able to sample a crowd as large as the one available to Dependabot, which presents a real issue with the concept of using the crowd to assess the risk of a dependency update. Therefore, Dependabot should make use of metrics other than the number of candidate and successful updates for a specific origin and target version of a provider package when attempting to support clients in dependency management, especially when the number of candidate updates is low.

Approach. Our goal is to explore alternative metrics that Dependabot could take into account to provide a sense of the compatibility of a new dependency version in a client package, particularly when the crowd does not provide enough support to calculate a compatibility score for the dependency update. We examine 7 metrics divided into two dimensions (*origin version range compatibility scores* and *client history of updates*) that can be calculated using data already available to Dependabot (summarized in Table 5.1).

Origin Version Range Compatibility Scores: Because Dependabot exclusively counts candidate updates with the same origin and target version of a provider towards a compatibility score, the number of candidate updates for each compatibility score is severely limited. While there may be a high number of candidate updates for the provider package overall, all of these candidate updates end up being spread across a wide range of potential origin version and target version combinations.

Table 5.1: Dimensions and their features that are used to assess the compatibility of a new dependency version in a client package when the crowd does not provide enough support to calculate a compatibility score for the dependency update.

Dimension	**Metric Name**	**Rational**	**Description**
Origin Version Range Compatibility Scores	Patch Origin Version Range Compatibility Score	Amplify the input from the crowd by considering candidate updates from similar origin versions.	The compatibility score for provider package P calculated using the candidate updates from all 3-tuples matched in (P, `x.y.*`, `x.y.z`).
	Minor Origin Version Range Compatibility Score	Amplify the input from the crowd by considering candidate updates from origin versions which may be less similar than patch ranges, but still should not contain breaking changes.	The compatibility score for provider package P calculated using the candidate updates from all 3-tuples matched in (P, `x.*.*`, `x.y.z`).
	Major Origin Version Range Compatibility Score	Maximize amplifying the input from the crowd by incorporating candidate updates from all origin versions.	The compatibility score for provider package P calculated using the candidate updates from all 3-tuples matched in (P, `*.*.*`, `x.y.z`).
Client History of Updates	Passing Dependabot PRs	Captures the historical stability of the client's package in general.	The number of Dependabot PRs previously opened in the client package that have successfully passed the client's CI pipeline.
	Passing Provider Dependabot PRs	Captures the historical stability between the client package and the provider package Dependabot is opening a PR to update.	The number of Dependabot PRs for the same provider previously opened in the client's package that have successfully passed the client's CI pipeline.
	Merged Dependabot PRs	Captures the level of trust the client package has with Dependabot PRs in general.	The number of Dependabot PRs previously merged by a user in the client's package.
	Merged Provider Dependabot PRs	Captures the level of trust the client package has in the provider package Dependabot is opening a PR to update.	The number of Dependabot PRs for the same provider previously merged by a user in the client's package.

To address this issue, we consider using the candidate updates from a range of origin versions that have been updated to a specific target version. We take inspiration from the Semantic Versioning (SemVer) scheme[16], a popular policy for communicating the type of changes made to a software package, where clients can specify whether they would like to accept a range of versions from the provider[17] (Dietrich et al., 2019; Decan and Mens, 2020). Similarly, we calculate three origin version range compatibility score metrics: a *patch origin version range compatibility score*, a *minor origin version range compatibility score*, and a *major origin version range compatibility score*. These origin version range compatibility scores are calculated in the same way as the raw compatibility score (i.e., the number of successful updates divided by the number of candidate updates), but they each consider an increasingly wider range of origin versions of the provider package to select candidate updates from. The patch origin version range compatibility score considers all origin versions of a provider package where only the patch version number of the origin version may differ. The minor origin version range compatibility score considers all origin versions of a provider package where the minor or patch version numbers of the origin version may differ. Finally, the major origin version range compatibility score considers all origin versions of a provider package where the major, minor or patch version numbers of the origin version may differ (i.e., all origin versions of a provider package that have been updated to a specific target version). Table 5.1 provides an example of these matching patterns. It can be seen that the major origin version range compatibility score will match more 3-tuple updates than the minor

[16] `https://semver.org`
[17] `https://nodesource.com/blog/semver-tilde-and-caret/`

origin version range compatibility score, which in turn will match more 3-tuple updates than the patch origin version range compatibility score. These metrics aim to amplify the input from the crowd by expanding the range of considered candidate updates for each compatibility score at the cost of generalizing the exact origin version of the provider being considered for each origin version range compatibility score.

Client History of Updates: Even when considering the candidate updates from a range of origin versions, there still might not be enough support from the crowd to reliably calculate a compatibility score. Therefore, we turn to historical metrics from the client package to help assess the involved risk with a dependency update. Specifically, we look at the number of Dependabot PRs previously opened that passed the CI pipeline in the client package (both overall and for each specific dependency). These metrics aim to capture the historical stability of the client's package in general, as well as the historical compatibility between the client package and the provider package that the Dependabot PR is attempting to update.

Additionally, we consider the number of Dependabot PRs that have previously been merged in the client package (both overall and for each specific dependency). These metrics aim to capture the level of trust the client has in the specific provider package Dependabot is attempting to update and the client's overall providers in general, as a higher number of merged Dependabot PRs suggests a higher level of trust by the client.

To investigate how well the individual dimensions can assess the compatibility of a dependency (i.e., their predictive power), we built a random forest model for each of the previously discussed dimensions, setting the dependent variable as whether the Dependabot PR is merged by a client package developer. We select whether the client package developer accepts or rejects the dependency update rather than, for example, the result of the client's CI pipeline running against the dependency update, because it is common for CI pipelines to contain low-quality tests (Hejderup and Gousios, 2021), which would bias how we assess the compatibility of the dependency update (e.g., low-quality CI pipelines might often fail due to flaky tests). Using whether a client package developer decides to merge the Dependabot PR allows us to capture whether deliberate action was taken by a human to either accept or reject the dependency update, as client package developers may be aware of issues with their CI pipeline and merge Dependabot PRs with failed CI pipelines anyway because they know their pipeline failed for reasons unrelated to the dependency update. In fact, we found this to be the case in 28% of Dependabot PRs with a failed CI pipeline, where the client decides to merge the Dependabot PR anyway.

When building these models, we only consider Dependabot PRs from the 4-tuple dataset that have fewer than 5 candidate updates, as these are the cases where we have recorded the Dependabot PR and the crowd has not provided enough support for Dependabot to calculate a reliable compatibility score. As a baseline, we build a random forest model with the raw compatibility score as the sole independent variable and the dependent variable as the merge result of the Dependabot PRs. For our baseline model, we only consider Dependabot PRs from the 4-tuple dataset set that have at least 5 candidate updates. We use the `ranger`[18] package in `R` as our random forest implementation due to its enhanced performance.

[18] https://cran.r-project.org/web/packages/ranger/ranger.pdf

To validate the performance and stability of our built models, we performed 100 out-of-sample bootstrap iterations to compute the median AUC (Area Under the receiver operator characteristics Curve) for each model. Prior work (Tantithamthavorn et al., 2017; Lee et al., 2020) has shown that the out-of-sample bootstrap technique had the best balance between the bias and variance of estimates. The out-of-sample bootstrap technique randomly samples data with replacement for n iterations. The sampled data in an iteration is used as the training set for that iteration, while the data that was not sampled in that iteration is used as the testing set for that iteration. We then trained a model with the training set and calculated the AUC of the model with the testing set for each iteration.

In addition, to investigate how well both of the studied dimensions can help to assess the compatibility of a dependency, we built a random forest model using all 7 metrics from both dimensions previously discussed. We evaluated the performance of this combined model using the same aforementioned process of computing the median AUC of the model with 100 out-of-sample bootstrap iterations.

Findings. **Observation 5.3)** ***There is room for improvement when establishing the relationship between the compatibility score for a dependency update and whether the associated Dependabot PR is merged by the client package.*** We observe that our baseline model built with the compatibility score as the sole predictor variable only achieves a median AUC of 0.62 (Figure 5.6 shows the distribution of AUC improvements compared to the median AUC for this baseline model). This shows that there is room for improvement when establishing the relationship between the compatibility score and the result of whether the client merged the Dependabot PR, and that it could be beneficial for Dependabot to consider further metrics when trying to convey how compatible a dependency update really is for client packages.

Observation 5.4) ***Considering a range of origin versions for a specific target version can help increase the number of candidate updates used to calculate the compatibility score.*** While we find that the majority of compatibility scores from the 3-tuple dataset do not see any increase in the number of candidate updates used to calculate a compatibility scores when considering the patch origin version range (although the third quartile see 3x the number of candidate updates), the minor and major origin version range compatibility scores are able to consider respectively 5x and 10x the number of candidate updates as what is used by the raw compatibility score. We see relatively smaller improvements in the 4-tuple dataset, with the patch, minor, and major origin version range compatibility scores seeing respectively a median of 1x, 1.5x, and 1.9x the number of candidate updates as what is used by the raw compatibility score. Figure 5.5 shows the distribution of ratios of candidate updates for each origin version range compatibility score to the associated original compatibility score from the 3-tuple and 4-tuple datasets.

Considering a range of origin versions for a specific target version can also help to increase the number of compatibility scores that meet the required threshold number of candidate updates (i.e., 5) for Dependabot to display the badge on the associated PR. Recall from RQ1 that only 17% of compatibility scores from the 3-tuple dataset have at least 5 candidate updates, while 83% of compatibility scores from the 4-tuple dataset have at least 5 candidate updates. When we consider our calculated origin version range compatibility scores for the 3-tuple dataset, we find that 39%, 68%, and 78% of patch, minor, and major origin version range compatibility scores respectively have at least 5 candidate updates. For the 4-tuple dataset, we find that 86%, 90%, and 92% of patch, minor, and major origin version range compatibility scores respectively have at least 5 candidate updates.

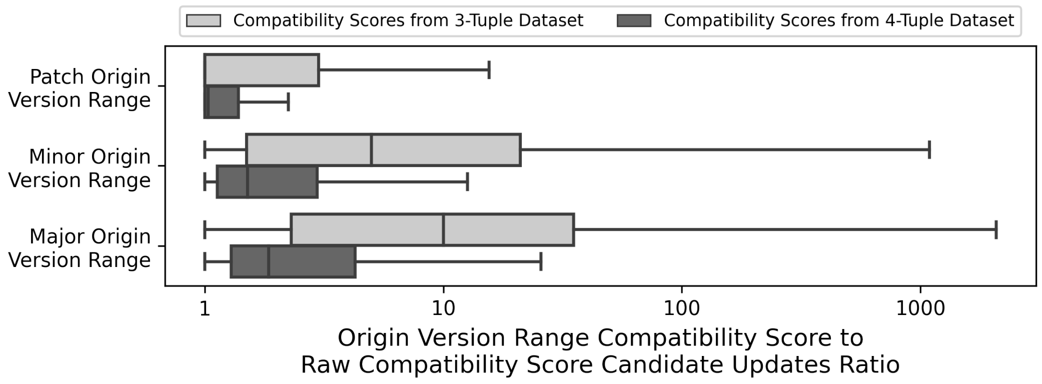


Figure 5.5: The distribution of ratios of candidate updates for each origin version range compatibility score to the associated raw compatibility score from the 3-tuple and 4-tuple datasets.

Observation 5.5) ***Both the origin version range compatibility scores and the client history of updates dimensions have significant predictive power to assess whether a client package developer will accept or reject a dependency update.*** The origin version range compatibility scores model achieves a median AUC 2.4% higher (0.64) than the base model, with the minor origin version range compatibility score having the highest permutation importance. The client history of updates model performs even better, achieving a median AUC 21.5% higher (0.76), with the number of Dependabot PRs previously merged in the client package having the highest permutation importance. Figure 5.6 shows the distribution of AUC improvements of both of these models compared to the baseline model median AUC. Figure 5.7 and Figure 5.8 show the distribution of permutation importance's of each metric for the origin version range compatibility scores model and the client history of updates model, respectively.

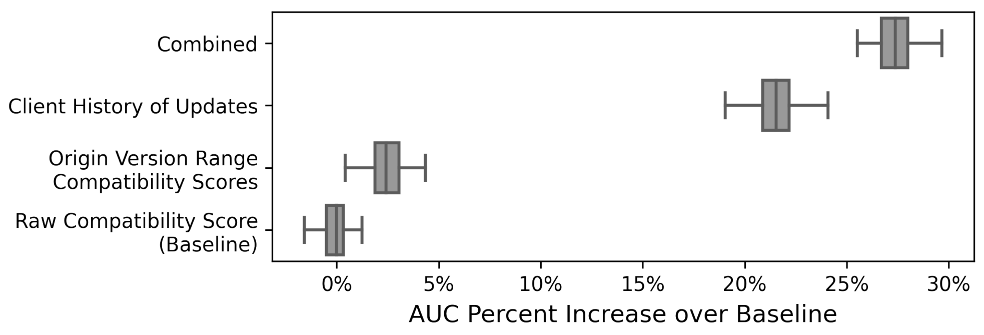


Figure 5.6: The distribution of the improvement in AUCs of models constructed with an individually studied dimension, and with all studied dimensions combined, compared against the baseline model.

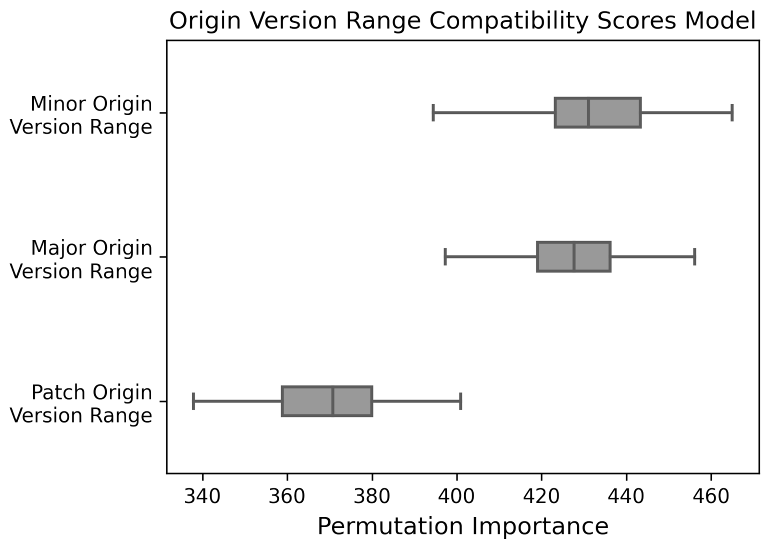


Figure 5.7: The distribution of permutation importance's of each metric from the origin version range compatibility scores model.

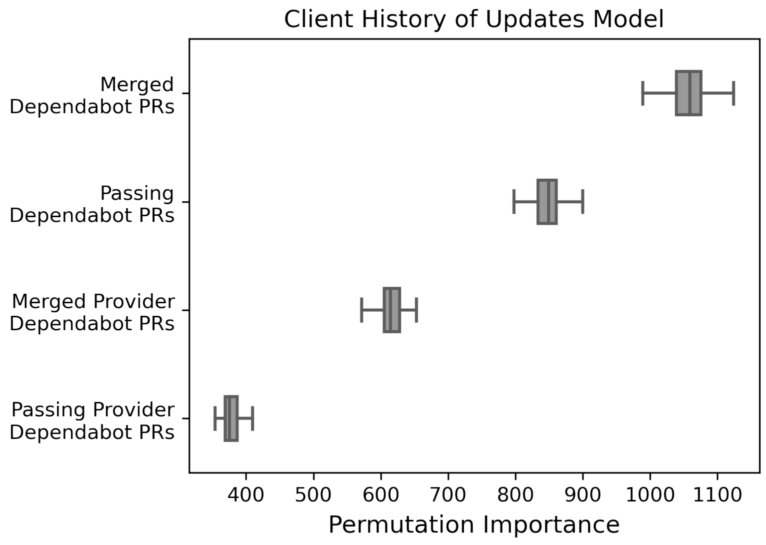


Figure 5.8: The distribution of permutation importance's of each metric from the client history of updates model.

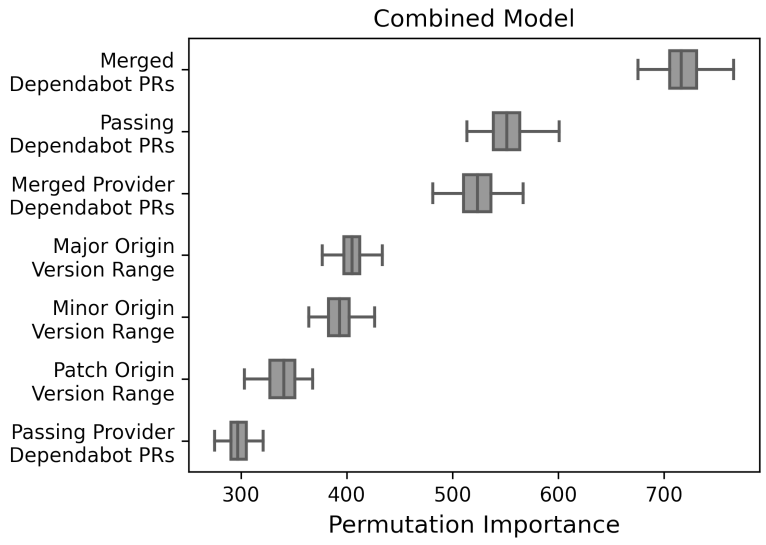


Figure 5.9: The distribution of permutation importance's of each metric from the model with combined metrics from both dimensions.

Observation 5.6) ***Combining metrics from both dimensions into a single model results in a larger predictive power than each of the studied dimensions individually.*** Figure 5.6 shows the distribution of AUC improvements of the model that combines the metrics from both the origin version range compatibility scores dimension and the client history of updates dimension compared to the median AUC of the baseline model. It can be seen that the combined model has a median AUC 27.4% higher (0.80) than the base model, which is 0.18 points higher than the base model, 0.16 points higher than the origin version range compatibility scores model, and 0.02 points higher than the client history of updates model. Figure 5.9 shows the distribution of permutation importances of each metric for the model with combined metrics from both dimensions.

RQ2: Which other sources of information can be considered when the crowd does not provide enough support to calculate a compatibility score?

- Considering a range of origin versions to a specific target version helps to increase the number of candidate updates, effectively amplifying the knowledge from the crowd.
- Metrics from the origin version range compatibility scores and client history of updates dimensions can help improve the prediction of whether a dependency update will be merged by a client package developer, with the model combining all metrics having the highest performance.
- Historical upgrade metrics from a client package tend to have the highest predictive power when considering whether said client package will accept or reject a dependency update.

5.4.3 RQ3: How much confidence should client packages place in the compatibility score?

Motivation. As previously explained, the compatibility score for a 3-tuple dependency update is calculated as the ratio of successful updates to the total number of candidate updates. The problem here is that a compatibility score with 5 successful candidate updates (i.e., 100%) will result in the associated dependency update appearing as more compatible than a dependency update with a compatibility score consisting of 99 successful updates and 1 failed update (i.e., 99%). But clearly, the latter dependency update is more likely to result in further successful updates. We use the phrase "more likely" since it is possible that the dependency update with the former compatibility score consisting of 5 successful updates is in fact more compatible in other client packages than the latter with the compatibility score consisting of 99 successful updates. The hesitation to agree with this hypothesis is because we have not seen the other 95 potential candidate updates that would contribute to the compatibility score of the former dependency update. Perhaps it will achieve an additional 95 successful updates and 0 failed updates and be considered better than the latter, though not likely.

Not only does the quantity of candidate updates affect how much confidence client packages should place in the compatibility score, but also the quality of these candidate updates. For example, if a Dependabot PR is contributing as a candidate update to a compatibility score, but the client's CI pipeline only consists of a linter check, it will bear the same weight as a candidate update with which the associated client's CI pipeline consisting of a build check, unit & integration test checks, and a deployment check. Evidently, client packages would be more inclined to place a higher level of confidence (i.e., trust) in a compatibility score that was calculated using candidate updates similar to the latter example rather than the former. Therefore, in this RQ, we investigate 1) how trustworthy are the compatibility scores based on the quantity of candidate updates? and 2) how trustworthy are the compatibility scores based on the quality of candidate updates?

Approach. Our goal is to explore how much confidence (i.e., trust) client packages should place in the compatibility scores. Our approach takes into account the quantity of candidate updates to calculate a metric that helps calibrate the trust of client packages in compatibility scores. Our metric is a confidence interval that is based on the total number of candidate and successful dependency updates used to calculate the score, which is the ratio of successful updates to candidate updates for a dependency update. The higher the number of candidate updates that are used to calculate the compatibility score, the more confident we are in this compatibility score. We calculate a 90% confidence interval for each compatibility score based on the approach described by Davidson-Pilon (2015) (further details are given in Appendix C). We choose a 90% confidence level because it is the least strict of the three most commonly used confidence levels (i.e., 90%, 95%, and 99%), as we aim to help clients estimate the compatibility of a dependency update, which does not require exact measurements as is the case in other, more critical situations (e.g., dealing with human life) (Hazra, 2017).

To explore the trustworthiness of the compatibility scores from the standpoint of quantity of candidate updates, we examine the distribution of confidence intervals across both the 3-tuple and 4-tuple datasets, as well as the distance from the compatibility score to the furthest confidence interval bound (i.e., the confidence interval precision). To explore the trustworthiness of the compatibility scores from the standpoint of quality of candidate updates, we examine the number and types of checks that ran against Dependabot PRs in our 4-tuple dataset, which were classified in Section 5.3.2. We exclusively examine Dependabot PRs that had a previously passing test suite (i.e., the CI pipeline has a successful conclusion on the main branch the Dependabot PR is based off), which is a requirement for Dependabot to consider the PR as a candidate update.

Findings. **Observation 5.7)** ***The confidence Dependabot has in compatibility scores can vary wildly, even though they are always presented to client packages as a similar badge.*** We find that half of compatibility scores with at least 5 candidate updates have a 90% confidence interval precision (i.e., the distance from the compatibility score to the furthest CI bound, see Appendix C - Equation 2) greater than 15%. The precision improves when examining compatibility scores from the 4-tuple dataset, with the median confidence interval precision dropping to 3.5%. This improvement is expected, as compatibility scores from the 4-tuple dataset tend to have more candidate updates than those in the 3-tuple dataset. This shows that, while Dependabot will present every compatibility score as a similar badge on PRs, it is very common for the confidence Dependabot has in these scores to vary wildly. This can mislead client packages into thinking the dependency update is in fact more stable than it actually is. The distributions of the 90% confidence interval precision for both the 3-tuple and 4-tuple datasets are shown in Figure 5.10.

Observation 5.8) ***CI pipelines for candidate update Dependabot PRs often contain a mixture of check types that are not always helpful for testing the compatibility of the dependency update.*** We find that candidate update Dependabot

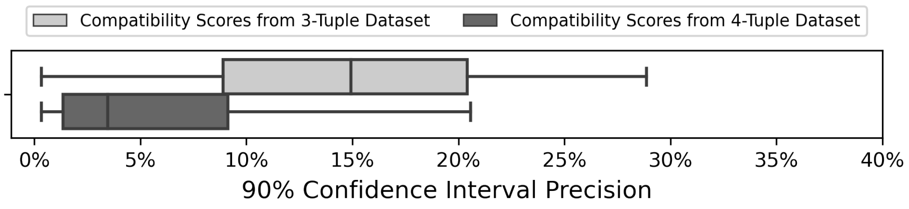


Figure 5.10: The distributions of the 90% confidence interval precision for both the 3-tuple and 4-tuple datasets.

PRs have a median of 3 checks that make up the client's CI pipeline to test the dependency update. The vast majority (94%) have at least a build or test check that is part of the CI pipeline. However, while these checks may seem promising, it is worth noting that it is common for client's tests to not thoroughly exercise the package's dependencies (Hejderup and Gousios, 2021). Additionally, recall that just over 1 in 10 (11%) check runs we collected were considered to be useless from the standpoint of contributing to the compatibility score. A quarter (26%) of candidate update Dependabot PRs have a CI pipeline that contains at least one of these useless check, while 1% of candidate update Dependabot PRs have only useless checks that ran against the dependency update. Of the candidate update Dependabot PRs with only useless checks, 94% of them had a successful CI conclusion, compared with 88% of Dependabot PRs with at least one build check.

RQ3: How much confidence should client packages place in the compatibility score?

- Client packages should be hesitant to place total confidence in the accuracy of compatibility scores, as more than half of the scores with at least 5 candidate updates have a 90% confidence interval precision greater than 15%.
- Candidate updates that contribute to compatibility scores may not always truly test the associated dependency update.

5.5 Discussion

In this section, we discuss the findings observed in Section 5.4. We present a set of practical implications for designers of dependency management bots with the aim of using the crowd to help client packages assess the risk of accepting a dependency update.

Implication 5.1) ***When the crowd does not provide enough support to calculate a compatibility score for a specific dependency update, dependency management bots should consider candidate updates from different origin version ranges to amplify the input from the crowd.*** We found in RQ1 that fewer than 1 in 5 of dependency updates have at least 5 candidate updates, which is the threshold required for the compatibility score to be shown on Dependabot PRs. We hypothesize that such a low proportion of compatibility scores with less than 5 candidate updates can be partially explained by two reasons. First, while Dependabot may be opening a high number of PRs in client packages for dependency updates, only a small portion of these client packages actually meet the requirements set by Dependabot for these PRs to be considered as candidate updates that contribute to the associated compatibility score (i.e., the client has a CI pipeline configured and a previously passing test suite on the main branch). Second, while many client packages may use the same provider package as a dependency, Dependabot may not be creating PRs to update that provider from the same origin and target versions. For example, for the provider package P and the 3 versions V_1, V_2, and V_3 of P, the 3-tuple updates (P, V_1, V_3) and (P, V_2, V_3) will have two different compatibility scores with separate candidate updates. So, while there may be a potentially high number of candidate updates for the provider package overall, all of these candidate updates end up being spread across a wide range of potential origin version and target version combinations, resulting in a low number of candidate updates for each specific origin version and target version combination.

While exploring the idea of Dependabot considering candidate updates from a range of origin versions in RQ2, we found that this ratio increases to 2 in 5 when considering all candidate updates from dependency releases with only a different patch origin version number, and to over two-thirds when considering all candidate updates from dependency releases with only a different minor or patch origin version number. Moreover, while considering the candidate updates from the patch origin version range did not result in a significant increase for the majority of compatibility scores, we found that considering a minor origin version range resulted in 5x the number of candidate updates as the raw compatibility score, while considering a major origin version range resulted in 10x the number of candidate updates as the raw compatibility score. This is reflected in our results from RQ2, where we found that the minor and major origin version range compatibility scores have the highest permutation importance in the origin version range compatibility scores model.

These are significant improvements which can lead to a more general form of the compatibility score being available and useful to a higher number of client packages while attempting to minimize the accuracy lost due to the range of origin versions being considered for the score. The SemVer policy specifies that important backward compatible changes require an update of the minor version component, and backward compatible bug fixes require an update of the patch version component (Decan and Mens, 2020). Assuming this policy is followed by provider package maintainers, which Decan and Mens (2020) found is becoming more common as software ecosystems mature, considering patch and minor origin version ranges will still be able to provide a reasonably accurate compatibility score, as major origin version range compatibility scores may be biased by breaking updates purposely introduced by package maintainers, since the SemVer scheme specifies that backward incompatible changes require an update of the major version component. However, designers of dependency management bots should make it clear to client packages that the origin version range compatibility scores might not be representative of the exact dependency update the client package is considering.

Implication 5.2) ***When there are simply not enough candidate updates from the crowd, dependency management bots should consider historical update metrics from the client package.*** We found in RQ1 that 83% of compatibility scores do not have enough candidate updates to be shown on Dependabot PRs. We explored potential solutions to this issue in RQ2, one of which was to consider the candidate updates from a range of origin versions, which we discussed in the previous implication. However, there are still cases where there are simply not enough candidate updates from the crowd to calculate a trustworthy compatibility score, even when considering the origin version range scores. Specifically, more than half (61%) of patch origin version range compatibility scores still have fewer than 5 candidate updates.

In these situations, dependency management bots should turn to historical client upgrade metrics to help clients assess whether they should accept or reject a dependency update. Not only did this dimension result in the model with the highest median AUC (improvement of 21.5% over the baseline), but we also found that the number of Dependabot PRs previously merged by a client package developer and the number of Dependabot PRs previously passing the client's CI pipeline were the most important features in our combined model. This suggests that taking historical upgrade metrics from the client package into account when considering a dependency update could be an effective way to provide a personalized compatibility score for each client.

In fact, it can be beneficial for dependency management bots to take these additional metrics into account not only when input from the crowd is low, but also when input from the crowd is high. We tested our models on Dependabot PRs with at least 5 candidate updates, and found similar results as in RQ2, with the combined model achieving an AUC of 0.78, 0.16 points higher than the baseline model.

Implication 5.3) ***Regardless of the level of input from the crowd, dependency management bots should provide supporting metrics alongside compatibility scores to signal the level of trust client packages should place in the compatibility score.*** We found in RQ1 that it is common for compatibility scores to have a low number of candidate updates, and what is lacking from the compatibility score is supporting information that tells client packages how much they can trust the recommendation from Dependabot. When people interact with any complex system (e.g., software bots), they create a mental model, which facilitates their use of the system (Norman, 2002; Kulesza et al., 2012). In automation-supported software engineering (e.g., deciding whether to update a dependency), valid mental models of the reliability of the output (e.g., the compatibility score of the dependency update) help the user (e.g., a client package) to know when to trust the recommendation. In fact, Zhang et al. (2020) found that confidence intervals can help calibrate people's trust in automation-supported decision-making. Similarly, confidence intervals could help Dependabot's compatibility score by providing client packages with an estimate of its trustworthiness, so that clients are able to distinguish between a 100% compatibility score with only 5 candidate updates and a 99% compatibility score with 99 candidate updates.

We explore this idea of using a confidence interval to help calibrate the level of trust client packages should place in the compatibility score in RQ3. We find that if a 90% confidence interval based on the number of candidate updates used to calculate a compatibility score was included on Dependabot PRs, half would show that the confidence interval precision was greater than 15%. These findings suggest that the level of trust client packages should place in compatibility scores can vary wildly, even though the compatibility score is always presented as the same badge style on Dependabot PRs. Therefore, dependency management bots should include additional metrics, like the confidence interval we calculated in RQ3, that can help to calibrate the level of trust client packages should place in the compatibility score. Presenting this information could be especially useful to client packages that do not have a CI pipeline configured, as they stand to gain the most benefit out of leveraging the crowd to assess the risk of a dependency update, and therefore should be aware of the level of confidence with which the associated compatibility score has been calculated.

Implication 5.4) ***Regardless of the level of input from the crowd, dependency management bots should place higher weights on packages with high quality CI pipelines that thoroughly test the dependency being updated.*** We found in RQ3 that candidate updates that contribute to compatibility scores can have CI pipelines that contain a variety of different types of checks, and may not always test the associated dependency update. In the extreme scenario, we found that of the 1% of Dependabot PRs that only had useless checks run against the dependency update, 94% of them had a successful CI conclusion. So, while these Dependabot PRs may be contributing as successful updates towards the compatibility score, they have not tested the dependency at all.

To address this issue, dependency management bots should take into account additional metrics other than the simple pass/fail result of a client's CI pipeline, such as the types of tests and the level of test coverage in the client package, to ensure that only high-quality input from the crowd is being considered. We saw in RQ1 that compatibility scores are already heavily skewed toward the higher range, which may be influenced by the fact that too many of the candidate updates contributing to the scores are from low-quality pipelines in client packages that do not truly test the dependency update. Therefore, dependency management bots should attempt to quantify the level of quality of clients from the crowd, and then either only consider clients that truly test the dependency update (i.e., have a high-quality CI pipeline), or perhaps provide a weight to each client based on the level of quality of their CI pipeline. This idea is similar to that of "Security Scorecards"[19], in which a number of heuristics associated with software security are tested against a package's dependencies and assigned a score of 0-10. Dependency management bots

[19]https://github.com/ossf/scorecard

could apply the same principle against a package's CI pipeline, evaluating heuristics related to software build and test quality in order to assess whether they should consider the CI pipeline when evaluating the compatibility of a new provider package release. However, designers of dependency management bots should be mindful that, while this may lead to a higher quality compatibility score, the trade-off is that fewer candidate updates may be available to calculate the compatibility score.

5.6 Threats to Validity

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

5.6.1 Internal Validity

Threats to internal validity concerns factors that could have influenced our analysis and findings. The Dependabot API simply returns every compatibility scores Dependabot has a record of at the time of the request. This means that we only retrieve a single snapshot of the compatibility scores for the dependencies being updated, and that the scores we collected for specific updates might not be the actual scores that a practitioner saw when Dependabot opened a PR on their package. To investigate how this might have affected our analysis, we built a pipeline that runs three times per day (i.e., every 8 hours), and retrieves all the Dependabot PRs for the list of packages described in Section 5.3.1 that have been created since the previous pipeline run. We then extract the 3-tuple data for the provider package being updated, and retrieve the compatibility score for that update every time the pipeline runs for the next 14 days. If a different client has a Dependabot PR for that same 3-tuple update, the 14-day threshold will reset. We let the pipeline run from November 25, 2020 until April 1, 2021. This gives us a time series dataset of compatibility scores that allows us to explore how long it might take or how many candidate updates are required for a compatibility score to become stable. We consider a score to have become stable at the latest point in time when the score is within 5% of the final score record. We find that 85.6% of compatibility scores we collect have the first score and all subsequent scores varying within 5% of the final compatibility score. In other words, only 14.4% of the compatibility scores were not instantly stable, which suggests that the analysis we conducted in our study would not have been significantly impacted by the fact that we only collect a single snapshot of the compatibility scores.

Additionally, the Dependabot API will not return a record for a specific 3-tuple dependency update if Dependabot has not recorded any candidate updates for said 3-tuple. Consequently, all compatibility scores we analyze have at least 1 candidate update. This means that the proportion of dependency updates that have compatibility scores with at least 5 candidate updates is likely lower than what we observe, a point which we mention in RQ1, as we disregard any dependency updates that have 0 candidate updates.

Another concern is related to the conclusions drawn when we consider whether a dependency update on a Dependabot PR caused the client's CI pipeline to fail. We use the checks associated with each PR to determine whether the update failed the CI pipeline, but checks can fail for reasons unrelated to the dependency update (e.g., flaky tests, license issues, etc.). However, this would have minimal effect in the context of studying the compatibility score, as Dependabot does not take the type of check that failed on a Dependabot PR into account when considering the PR as a candidate update that contributes to the compatibility score.

Finally, it is important to note that the conclusions drawn when considering whether a client package merged a Dependabot PR may have been affected by the fact that factors other than the compatibility score can contribute to the client package's decision of whether to merge the Dependabot PR or not. For example, a client package may be very risk averse, resulting in very few merged Dependabot PRs, regardless of the compatibility signals provided by Dependabot. Still, we attempt to account for this threat by taking into account the historical upgrade metrics for each client package when performing this analysis.

5.6.2 External Validity

Threats to external validity concern the generalization of our technique and findings. Our study only focuses on the compatibility score implementation of Dependabot. Therefore, our results cannot be generalized to different implementation of leveraging knowledge from the crowd to provide insights about the risk of a newly released version of a provider package. For example, the Renovate bot has a feature similar to Dependabot's compatibility score called *Merge Confidence*[20] that identifies and flags undeclared breaking releases based on analysis of test and release adoption data. Renovate's *Merge Confidence* has unique features that might differ from our results with Dependabot's compatibility score. Still, to the best of our knowledge, Dependabot was the first to leverage the crowd for dependency management by providing a compatibility score for each dependency update. So, while

[20] https://docs.renovatebot.com/merge-confidence/

our results cannot be generalized, our discussion provides implications that can still apply to other bots that leverage the crowd to assess the risk of an update.

5.7 Chapter Summary

Today's software systems are rarely built with code written by a single developer, with client packages often making use of specific versions of provider packages in the form of dependency relationships. These dependency relationships come with the essential and risky task of keeping the client package's dependencies up-to-date. Dependabot, an automated dependency management bot, helps facilitate this process by automatically opening PRs in client packages to update a dependency when a new version of the provider is released, as well as providing a *compatibility score* for each dependency update. This compatibility score is shown as a badge on PRs opened by Dependabot, and is meant to give clients a sense of the involved risk when updating a dependency by leveraging the knowledge of "the crowd", so that clients can be confident a new provider version is backwards compatible and bug-free.

In this chapter, we describe an empirical study of 579,206 Dependabot PRs, as well as 618,045 compatibility score records, that examines the viability of dependency management bots leveraging the crowd to help clients assess the involved risks with accepting a dependency update. We conclude that dependency management bots should go beyond only considering the result of clients' CI pipeline running against a dependency update when using the crowd to assess the risk of said dependency update. This is especially relevant since we found that the majority of compatibility scores do not have at least 5 candidate updates, which is the threshold required for the compatibility score badge to be displayed on Dependabot PRs, and when compatibility scores do have enough candidate updates, the vast majority of the scores are above 90%. As a result of this skewness in both the number of candidate updates and the scores themselves, dependency management bots should employ further methods to help amplify the input from the crowd or consider historical upgrade metrics to assess whether a client package should accept or reject a dependency update. Additionally, supporting metrics, such as a confidence interval, should be provided alongside the compatibility score to help calibrate the level of trust client packages should place in the score.

CHAPTER 6

Conclusions and Future Work

As today's software systems are increasingly built by client packages making use of provider packages in the form of dependency relationships, it becomes more important for these client packages to effectively manage their dependencies. By keeping their provider packages up-to-date, client packages can benefit from bug fixes, new functionalities, and security enhancements as they are released by the provider packages. However, this can prove to be a risky task, as new releases of the provider package may break API-backwards compatibility and introduce regressions into the client package. To help facilitate this dependency management process, clients are increasingly adopting dependency management bots to alert them when a provider they depend on releases a new version and whether the new version of the provider is compatible with their package.

In this thesis, we study the overhead that is introduced in clients that adopt these dependency management bots, as well as explore the viability of dependency management bots leveraging “the crowd” to help clients assess the risks that are involved with accepting a dependency update. Based on our studies, we provide a series of practical recommendations 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, as well as how bots can effectively leverage the crowd to further aid clients with dependency management.

6.1 Thesis Contributions

The main contributions of our thesis are listed below.

1. This is the first work to perform a large empirical study on the overhead that is introduced in client projects who adopt dependency management bots. We report that, while dependency management bots 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 practical recommendations to help designers of dependency management bots to reduce the amount of unnecessary work they create in client packages.

2. This is the first work to study on a large scale the efficacy of dependency management bots leveraging the crowd to provide supporting metrics to help clients assess the risk of accepting a dependency update. We report that dependency management bots should be mindful of ways to amplify the input

from the crowd, consider historical metrics from the client package to provide a personalized compatibility score, and include a confidence interval to help calibrate the trust clients should place in these crowd-sourced metrics. Our findings will help designers of dependency management bots effectively leverage crowd-sourced data to aid client packages with dependency management.

6.2 Future Work

In the following, we list avenues for future research that can be leveraged from our results.

- *Further research should be done into how dependency management bots should implement features specifically tailored to help reduce the amount of overhead introduced on the client.* For example, none of the bots mentioned throughout this thesis 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, in order to help catch false positive breaking updates. Additionally, these bots do not provide any form of incentive to resolve breaking dependency updates, 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 to client developers, as well as ensuring that crowd-source compatibility indicators are not biased by failed client pipelines that were broken by issues unrelated to the dependency update.

- *Further research should be done on exploring the efficacy of 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. (2020)), and transform those parts of the code to become compatible with the new provider version (such as in Nielsen et al. (2021)).* Combining these areas of research could prove to be an effective approach to reducing the effort on the part of client developers to address 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. Additionally, these methods could provide a compatibility score for each dependency update that is extremely personalized to each client package, as they could potentially detect whether a client package is affected by a non-backwards compatible API changes in the provider package.

- *Further research should be done to explore the true amount of work that is required to address breaking dependency issues.* While in Chapter 4 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 breaking dependency issues. However, this is not always easily measured, as it is difficult to accurately quantify the amount of work required on the developer's part to address breaking dependency issues, 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 (Marks et al., 2011). 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.

Bibliography

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017a). Why do developers use trivial packages? An empirical case study on npm. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*, pages 385–395.

Abdalkareem, R., Oda, V., Mujahid, S., and Shihab, E. (2020). On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empirical Software Engineering*, 25(2):1168–1204.

Abdalkareem, R., Shihab, E., and Rilling, J. (2017b). What Do Developers Use the Crowd For? A Study Using Stack Overflow. *IEEE Software*, 34(2):53–60.

Alfadel, M., Costa, D. E., Shihab, E., and Mkhallalati, M. (2021). On the Use of Dependabot Security Pull Requests. In *Proceedings of the 18th International Conference on Mining Software Repositories (MSR '21)*, pages 254–265.

Armstrong, R. A. (2014). When to use the Bonferroni correction. *Ophthalmic and Physiological Optics*, 34(5):502–508.

Barua, A., Thomas, S. W., and Hassan, A. E. (2014). What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empirical Software Engineering*, 19(3):619–654.

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116.

Bauer, D. F. (1972). Constructing Confidence Sets Using Rank Statistics. *Journal of the American Statistical Association*, 67(339):687–690.

Bogart, C., Kastner, C., and Herbsleb, J. (2015). When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies. In *Proceedings of the 30th International Conference on Automated Software Engineering Workshop (ASEW '15)*, pages 86–89.

Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. (2016). How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE '16)*, pages 109–120.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the Factors that Impact the Popularity of GitHub Repositories. In *Proceedings of the 2016 International Conference on Software Maintenance and Evolution (ICSME '16)*, pages 334–344.

Brito, A., Valente, M. T., Xavier, L., and Hora, A. (2020). You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25(2):1458–1492.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018a). APIDiff: Detecting API breaking changes. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, pages 507–511.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018b). Why and how Java developers break APIs. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*, pages 255–265.

Brown, C. and Parnin, C. (2019). Sorry to Bother You: Designing Bots for Effective Recommendations. In *Proceedings of the 1st International Workshop on Bots in Software Engineering (BotSE '19)*, pages 54–58.

Brown, C. and Parnin, C. (2020). Sorry to Bother You Again: Developer Recommendation Choice Architectures for Designing Effective Bots. In *Proceedings of the 42nd International Conference on Software Engineering Workshops (ICSEW '20)*, pages 56–60.

Chowdhury, M. A. R., Abdalkareem, R., Shihab, E., and Adams, B. (2021). On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *IEEE Transactions on Software Engineering*.

Cliff, N. (1996). *Ordinal methods for behavioral data analysis.* Lawrence Erlbaum Associates, Inc.

Cogo, F. R., Oliva, G. A., and Hassan, A. E. (2019). An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Transactions on Software Engineering*, 47(11):2457–2470.

Cohen, J. (1960). A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement*, 20(1):37–46.

Cox, J., Bouwers, E., Eekelen, M. v., and Visser, J. (2015). Measuring Dependency Freshness in Software Systems. In *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE '15)*, pages 109–118.

Davidson-Pilon, C. (2015). *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference*. Addison-Wesley Professional, 1st edition. pages 111–122.

Decan, A. and Mens, T. (2020). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240.

Decan, A., Mens, T., and Claes, M. (2017). An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*, pages 2–12.

Decan, A., Mens, T., and Constantinou, E. (2018a). On the Evolution of Technical Lag in the npm Package Dependency Network. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME '18)*, pages 404–414.

Decan, A., Mens, T., and Constantinou, E. (2018b). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the*

15th International Conference on Mining Software Repositories (MSR '18), pages 181–191.

Dietrich, J., Pearce, D., Stringer, J., Tahir, A., and Blincoe, K. (2019). Dependency Versioning in the Wild. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*, pages 349–359.

DiStaso, M. W. and Bortree, D. S. (2012). Multi-method analysis of transparency in social media practices: Survey, interviews and content analysis. *Public Relations Review*, 38(3):511–514.

Drouhard, M., Chen, N.-C., Suh, J., Kocielnik, R., Pena-Araya, V., Cen, K., Xiangyi Zheng, and Aragon, C. R. (2017). Aeonium: Visual analytics to support collaborative qualitative coding. In *Proceedings of the Pacific Visualization Symposium (PacificVis '17)*, pages 220–229.

Erlenhov, L., Gomes de Oliveira Neto, F., Scandariato, R., and Leitner, P. (2019). Current and Future Bots in Software Development. In *Proceedings of the 1st International Workshop on Bots in Software Engineering (BotSE '19)*, pages 7–11.

Erlenhov, L., Neto, F. G. d. O., and Leitner, P. (2020). An empirical study of bots in software development: characteristics and challenges from a practitioner's perspective. In *Proceedings of the 28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (SEC/FSE '20)*, pages 445–455.

Fard, A. M. and Mesbah, A. (2017). JavaScript: The (Un)Covered Parts. In *Proceedings of the 2017 International Conference on Software Testing, Verification and Validation (ICST '17)*, pages 230–240.

Fleiss, J. L. and Cohen, J. (1973). The Equivalence of Weighted Kappa and the Intraclass Correlation Coefficient as Measures of Reliability. *Educational and Psychological Measurement*, 33(3):613–619.

Foo, D., Chua, H., Yeo, J., Ang, M. Y., and Sharma, A. (2018). Efficient static checking of library updates. In *Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, pages 791–796.

Gallaba, K. and McIntosh, S. (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):33–50.

Gonzalez-Barahona, J. M., Sherwood, P., Robles, G., and Izquierdo, D. (2017). Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *Open Source Systems: Towards Robust Practices*, volume 496, pages 182–192. Springer International Publishing.

Gupta, A. K. (2011). Beta Distribution. In *International Encyclopedia of Statistical Science*, pages 144–145. Springer Berlin Heidelberg.

Gupta, A. K. and Nadarajah, S. (2004). *Handbook of Beta Distribution and Its Applications*. CRC Press.

Hazra, A. (2017). Using the confidence interval confidently. *Journal of Thoracic Disease*, 9(10):4125–4130.

Hejderup, J. and Gousios, G. (2021). Can we trust tests to automate dependency updates? A case study of java projects. *Journal of Systems and Software*, 183:111097.

Hespanhol, L., Vallio, C. S., Costa, L. M., and Saragiotto, B. T. (2019). Understanding and interpreting confidence and credible intervals around effect estimates. *Brazilian Journal of Physical Therapy*, 23(4):290–301.

Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE '16)*, pages 426–437.

Jafari, A. J., Costa, D. E., Abdalkareem, R., Shihab, E., and Tsantalis, N. (2020). Dependency Smells in JavaScript Projects. *arXiv:2010.14573 [cs]*.

Jezek, K., Dietrich, J., and Brada, P. (2015). How Java APIs break – An empirical study. *Information and Software Technology*, 65:129–146.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*, pages 92–101.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and Evolution of Package Dependency Networks. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*, pages 102–112.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018). Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration. *Empirical Software Engineering*, 23(1):384–417.

Kulesza, T., Stumpf, S., Burnett, M., and Kwan, I. (2012). Tell me more? the effects of mental model soundness on personalizing an intelligent agent. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI '12)*, pages 1–10.

Landis, J. R. and Koch, G. G. (1977). The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159.

LaToza, T. D. and van der Hoek, A. (2016). Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *IEEE Software*, 33(1):74–80.

Lebeuf, C., Zagalsky, A., Foucault, M., and Storey, M.-A. (2019). Defining and Classifying Software Bots: A Faceted Taxonomy. In *Proceedings of the 1st International Workshop on Bots in Software Engineering (BotSE '19)*, pages 1–6.

Lebeuf, C. R. (2018). *A Taxonomy of Software Bots: Towards a Deeper Understanding of Software Bot Characteristics*. PhD thesis, Unversity of Victoria.

Lee, D., Rajbahadur, G. K., Lin, D., Sayagh, M., Bezemer, C.-P., and Hassan, A. E. (2020). An empirical study of the characteristics of popular Minecraft mods. *Empirical Software Engineering*, 25(5):3396–3429.

Li, L., Bissyandé, T. F., Wang, H., and Klein, J. (2018). CiD: automating the detection of API-related compatibility issues in Android apps. In *Proceedings of the*

27th International Symposium on Software Testing and Analysis (ISSTA '18), pages 153–163.

Lim, W. (1994). Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30.

Lin, B., Zagalsky, A. E., Storey, M.-A., and Serebrenik, A. (2016). Why Developers Are Slacking Off: Understanding How Software Teams Use Slack. In *Proceedings of the 19th Conference on Computer Supported Cooperative Work and Social Computing Companion (CSCW '16)*, pages 333–336.

Marks, L., Zou, Y., and Hassan, A. E. (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)*, Promise '11, pages 1–8.

McDonald, N., Schoenebeck, S., and Forte, A. (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):1–23.

Mezzetti, G., Møller, A., and Torp, M. T. (2018). Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP '18)*.

Mileva, Y. M., Dallmeier, V., Burger, M., and Zeller, A. (2009). Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE-Evol '09)*, pages 57–62.

Mirhosseini, S. and Parnin, C. (2017). Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE '17)*, pages 84–94.

Mohagheghi, P., Conradi, R., Killi, O. M., and Schwarz, H. (2004). An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 282–292.

Mujahid, S., Abdalkareem, R., Shihab, E., and McIntosh, S. (2020). Using Others' Tests to Identify Breaking Updates. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*, pages 466–476.

Munson, J. and Elbaum, S. (1998). Code churn: a measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*, pages 24–31.

Møller, A., Nielsen, B. B., and Torp, M. T. (2020). Detecting locations in JavaScript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25.

Møller, A. and Torp, M. T. (2019). Model-based testing of breaking changes in Node.js libraries. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, pages 409–419.

Nagappan, N. and Ball, T. (2005). Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 284–292.

Nielsen, B. B., Torp, M. T., and Møller, A. (2021). Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*, pages 74–85.

Norman, D. A. (2002). *The design of everyday things*. Basic Books.

Peng, Z., Yoo, J., Xia, M., Kim, S., and Ma, X. (2018). Exploring How Software Developers Work with Mention Bot in GitHub. In *Proceedings of the Sixth International Symposium of Chinese CHI (ChineseCHI '18)*, pages 152–155.

Raemaekers, S., van Deursen, A., and Visser, J. (2014). Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM '14)*, pages 215–224.

Raemaekers, S., van Deursen, A., and Visser, J. (2017). Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 129:140–158.

Romano, J. and Kromrey, J. (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?

Rosen, C. and Shihab, E. (2016). What are mobile developers asking about? A large scale study using stack overflow. *Empirical Software Engineering*, 21.

Sim, J. and Wright, C. C. (2005). The Kappa Statistic in Reliability Studies: Use, Interpretation, and Sample Size Requirements. *Physical Therapy*, 85(3):257–268.

Storey, M.-A. and Zagalsky, A. (2016). Disrupting developer productivity one bot at a time. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE '16)*, pages 928–931.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2017). An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering*, 43(1):1–18.

Treude, C., Barzilay, O., and Storey, M.-A. (2011). How do programmers ask and answer questions on the web?: NIER track. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pages 804–807.

Urli, S., Yu, Z., Seinturier, L., and Monperrus, M. (2018). How to design a program repair bot? insights from the repairnator project. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEET '18)*, pages 95–104.

Vasilescu, B., Filkov, V., and Serebrenik, A. (2013). StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge. In *Proceedings of the 2013 International Conference on Social Computing (SOCIALCOM '13)*, pages 188–195.

Wessel, M. (2020). Enhancing developers' support on pull requests activities with software bots. In *Proceedings of the 28th Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), pages 1674–1677.

Wessel, M., de Souza, B. M., Steinmacher, I., Wiese, I. S., Polato, I., Chaves, A. P., and Gerosa, M. A. (2018). The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Proceedings of the ACM on Human-Computuer Interaction*, 2(182):1–19.

Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., and Gerosa, M. A. (2020a). Effects of Adopting Code Review Bots on Pull Requests to OSS Projects. In *Proceedings of the 2020 International Conference on Software Maintenance and Evolution (ICSME '20)*, pages 1–11.

Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., and Gerosa, M. A. (2020b). What to Expect from Code Review Bots on GitHub?: A Survey with OSS Maintainers. In *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES '20)*, pages 457–462.

Wessel, M. and Steinmacher, I. (2020). The Inconvenient Side of Software Bots on Pull Requests. In *Proceedings of the 42nd International Conference on Software Engineering Workshops (ICSEW '20)*, pages 51–55.

Wessel, M., Steinmacher, I., Wiese, I., and Gerosa, M. A. (2019). Should I Stale or Should I Close? An Analysis of a Bot That Closes Abandoned Issues and Pull Requests. In *Proceedings of the 1st International Workshop on Bots in Software Engineering (BotSE '19)*, pages 38–42.

Wessel, M., Wiese, I., Steinmacher, I., and Gerosa, M. A. (2021). Don't Disturb Me: Challenges of Interacting with Software Bots on Open Source Software Projects. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW2):1–21.

Wittern, E., Suter, P., and Rajagopalan, S. (2016). A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*, pages 351–361.

Wyrich, M., Ghit, R., Haller, T., and Müller, C. (2021). Bots Don't Mind Waiting, Do They? Comparing the Interaction With Automatically and Manually Created Pull Requests. In *Proceedings of the 3rd International Workshop on Bots in Software Engineering (BotSE '21)*, pages 6–10.

Wyrich, M., Hebig, R., Wagner, S., and Scandariato, R. (2020). Perception and Acceptance of an Autonomous Refactoring Bot. In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART '20)*, pages 303–310.

Zerouali, A., Constantinou, E., Mens, T., Robles, G., and González-Barahona, J. (2018). An Empirical Analysis of Technical Lag in npm Package Dependencies. In *New Opportunities for Software Reuse*, volume 10826, pages 95–110. Springer International Publishing.

Zhang, Y., Liao, Q. V., and Bellamy, R. K. E. (2020). Effect of confidence and explanation on accuracy and trust calibration in AI-assisted decision making. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency (FAT* '20)*, pages 295–305.

Appendices

A Patterns for classifying CI build types

The following steps are used to classify the checks that run as part of the client's CI pipeline on Dependabot PRs: i) the first author manually examined the 20 most popular unclassified check names (a check name is used to give a high-level description of the task the check performs) to extract patterns that could be grouped into similar categories, ii) these new patterns are added to a set of regular expressions that capture common check names and assign these checks to a specific overarching check type category, iii) the full data set of checks are then re-classified with the updated regular expressions, iv) the process is repeated using only checks that have not yet been classified until any new extracted patterns do not classify a threshold of at least 0.01% of the unclassified checks. Once this threshold was reached, 91.8% of the checks had been matched to one of six categories described in Table A.1.

B Distinguishing between the 3-tuple and 4-tuple datasets

The list of client projects (Section 5.3.1) that we collected Dependabot PRs for (Section 5.3.2) could have resulted in having multiple Dependabot PRs for the same 3-tuple update. For example, for the 3-tuple update (P, V_1, V_2) for updating the provider package P from version V_1 to V_2, we might have (C_1, P, V_1, V_2) and (C_2, P, V_1, V_2), where C_1 and C_2 are different client projects that make use of P as a dependency. However, we might not necessarily have at least one matching 4-tuple update for every 3-tuple update. In other words, our list of client projects built in Section 5.3.1 is by no means an exhaustive list of all projects that use Dependabot. So, if Dependabot opens a PR in client C for a 3-tuple update that is contributing as a candidate update, we would only potentially find a matching 4-tuple update for the associated 3-tuple update if C is in our list of client projects.

If a client C is in our list of projects, and they use provider package P as a dependency, there is no guarantee that Dependabot has opened a PR for every 4-tuple combination of (C, P, V_X, V_Y), where V_X and V_Y are two versions of P, with V_X being released prior to V_Y. For example, if C only adopts P as a dependency beginning at version V_4, then Dependabot would not have opened any PRs for the 4-tuple update (C, P, $V_{X<4}$, $V_{Y<4}$) where $V_{X<4}$ and $V_{Y<4}$ are versions of P that were released prior to V_4.

Additionally, Dependabot is constrained to opening PRs for dependency updates that fall within the client's dependency version specifications. For example, if C has version pinned P to version V_1, Dependabot would only open PRs that follow the 4-tuple (C, P, V_1, V_T), where V_T is the latest target release of P (assuming that C does not change their version specifications for P).

A final reason that might explain why Dependabot might not open a PR for specific 4-tuple updates is that clients are able to configure a limit for the number of Dependabot PRs open in their project at any one time[1]. So if the limit of PRs allowed to be opened by Dependabot at one time in client C has already been reached (e.g., updates for other dependencies), Dependabot will not create any new PRs to update P, even as P releases new versions, until the number of open Dependabot PRs in C has dropped below the limit.

As a concrete example to illustrate the distinction between the 3-tuple update and 4-tuple update datasets, we again use the scenario from Figure 5.1, which shows a Dependabot PR for the 4-tuple update (C, Husky, 6.0.0, 7.0.4). We determine that client C had integrated with Dependabot in Section 5.3.1. Then, the Dependabot PR shown in Figure 5.1 is collected in Section 5.3.2, where we determine that Dependabot has opened a PR to update the Husky provider package. Next, in Section 5.3.3, we collect the compatibility scores for Husky from the Dependabot API. We end up collecting the compatibility scores not only for the 3-tuple update (Husky, *6.0.0, 7.0.4*), but also all other 3-tuple combinations of (Husky, V_C, V_T) (e.g., (Husky, 6.0.1, 7.0.4), (Husky, 7.0.1, 7.0.4), (Husky, 6.0.0, 6.0.1), etc.). So, the compatibility scores for all of these 3-tuples we collected are included in the 3-tuple dataset. However, only the compatibility score for which Dependabot has opened a PR in C (e.g., (C, Husky, 6.0.0, 7.0.4)) is included in the 4-tuple dataset.

[1]https://docs.github.com/en/code-security/supply-chain-security/keeping-your-dependencies-updated-automatically/configuration-options-for-dependency-updates#open-pull-requests-limit

C Calculating a 90% confidence interval for a compatibility score

We approach the task of calculating a confidence interval for a compatibility score using Bayesian inference, which is a statistical approach that aims at estimating a certain parameter (e.g., a mean or a proportion) from the population distribution, given the evidence provided by the observed (i.e., collected) data (Hespanhol et al., 2019). In our case, we model the compatibility scores (i.e., the successful update ratio from the number of candidate updates) as a beta distribution, which defines random variables between 0 and 1, making it an ideal distribution choice for modelling the compatibility score (Gupta and Nadarajah, 2004). The beta distribution takes two parameters: a and b. We start with a beta prior with a=1 and b=1 (which is a uniform prior), and our observed data of successful and failed counts for a dependency update. For a given true successful update ratio p and N candidate updates, the number of successful updates S will look like a binomial random variable with parameters p and N, where N is the number of candidate updates and p is unknown. This is because of the equivalence between successful update ratio and probability of a candidate update being a success or failure, out of N possible candidate updates. So, with our Beta(a=1, b=1) prior on p and our observed successful updates $S \sim$ Binomial(N, p), then our posterior is also a beta distribution with $a = 1 + S$ and $b = 1 + N - S$.

We use a normal approximation to calculate the standard deviation of the posterior beta distribution (Gupta, 2011). That is,

$$\begin{aligned} \sigma &= \sqrt{\frac{ab}{(a+b)^2(a+b+1)}} \\ \text{where } a &= 1 + \textit{successful updates} \\ b &= 1 + \textit{failed updates} \end{aligned} \tag{1}$$

From there, we define a confidence interval precision as:

$$\textit{Precision}_{CI} = 1.65\sigma \tag{2}$$

where 1.65 is the critical (z) value (derived from the mathematics of the standard normal curve) to be used in confidence interval calculation associated with the 90% confidence level (Hazra, 2017).

We define the bounds of the 90% confidence interval as:

$$\begin{aligned} \textit{CI} = [&\textit{max}(CS - \textit{Precision}_{CI}, 0), \\ &\textit{min}(CS + \textit{Precision}_{CI}, 1)] \\ \text{where } CS = \; &\textit{the existing compatibility score} \end{aligned} \tag{3}$$

Figure C.1 shows the posterior distributions resulting from the aforementioned process for particular success/failure update pairs. It can be seen that the distributions with a lower number of total candidate updates (i.e., A and B) have relatively wide distributions, expressing the uncertainly about what the true successful update ratio might be, whereas the distributions with a higher number of candidate updates (i.e., C and D) have tighter distributions. The solid vertical lines in Figure C.1 show the original compatibility score for each particular success/fail update pair, while the dashed vertical lines show the bounds of the associated 90% confidence interval.

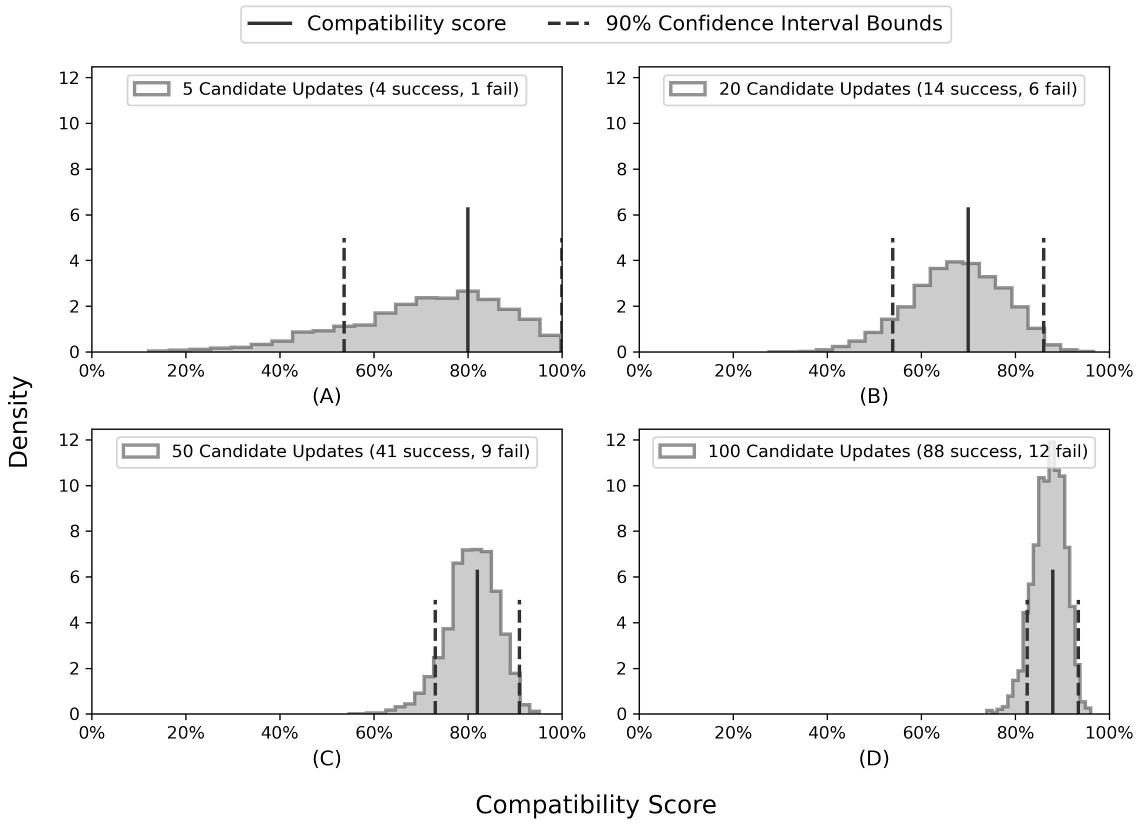


Figure C.1: Examples of posterior distributions for particular success/failure update pairs with relatively low (A and B) and high (C and D) candidate updates. The vertical lines mark the compatibility score (solid) and the upper and lower bounds of the associated 90% confidence interval (dashed).

Table A.1: String patterns for classifying the types of checks that ran against Dependabot PRs.

Category	Percent	Regular Expression (case-insensitive)
Build	58.1%	(^| |-)(build|install)|Travis CI|developing-with-angular|(main|workflow|setup)|Node(.js)? \\d?\\d?|(Continuous integration|^ci($\|))\|(tsc\|typescript)\|monica CI\|(web\|webpack)\|PHP\|(Try:)?ember((-\|)try)?\|(macOS\|windows\|ubuntu\|linux)(-latest)?\|Python\|^3.\\d$|^2.\\d$
Test	17.2%	(^|)test|Analy(s|z)e|Analysis|karma|e2e|stoplightio|check|unit-js| run|Validation|rspec
Useless	11.2%	WIP|^Rule: automatic merge for Dependabot pull requests \(merge\)$\|(Auto ?)?merge\|^stale$|^Update \.NET SDK$\|^Summary$|fixupbot|Mixed content|Rebase|Autosquash|Backport|docs|hyperjump|kodiakhq: status|DCO|lock|Discord Listener|Label|css|Clean GitHub pages|pre-commit|remove-pr|markdown-link-check|Run CircleCI artifacts redirector|pedrolamas.com|Auto Approve a PR by dependabot|dependabolt|github/dependabot.yml|greeting|chrome|firefox|finish|mui-org.material-ui|jbhannah.net|Always run job|jhipster.generator-jhipster|dispatch|Timeline protection|Inclusive Language|mark-duplicate|migration|Generate HTML log|feature flags
Lint	7.0%	(es)?lint|ESLint Report Analysis|codecov|Floating Dependencies|prettier| Coverage|Standard|bundle-size|pronto|flake8| mypy|CodeFactor|Code style
Deploy	4.9%	Redirect rules|Header rules|deploy|release|Pages changed|publish|artifact
Security Analysis	1.6%	code(| |-)ql|GitGuardian Security Checks|SonarCloud Code Analysis|LGTM analysis|depcheck|audit|rubocop