Leveraging the Crowd for Dependency Management: An Empirical Study on the Dependabot Compatibility Score

Benjamin Rombaut, Filipe R. Cogo, Ahmed E. Hassan

Abstract—Software is increasingly being built by client packages making use of third-party 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 client packages that use the provider package as a dependency. In this paper, our objective is to study the efficacy of leveraging the crowd to help client packages assess the risks involved 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 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 so as to avoid biasing the compatibility score.

Index Terms—Dependency Management, Software Bots, Crowd-sourcing, Mining Software Repositories, Dependabot

1 Introduction

Software is increasingly being built by making use of dependency relationships, where a client package relies on specific versions of a provider package. These provider packages enable code reuse and have been shown to improve developer productivity, software quality, and time-to-market of software products [1, 2]. 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 [3, 4].

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) [5]. 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 [6]. 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 [7].

However, many client packages do not have a full CI pipeline enabled [6], 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. [8] and Mezzetti et al. [9] 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[1] is an automated dependency management tool that packages on GitHub[2] can integrate with to au-

- *Benjamin Rombaut and Ahmed E. Hassan are with the Software Analysis and Intelligence Lab (SAIL) in the School of Computing at Queen's University, Kingston, Ontario, Canada.*
  Email: benjamin.rombaut@queensu.ca, ahmed@cs.queensu.ca
- *Filipe R. Cogo is with the Centre for Software Excellence (CSE) at Huawei, Canada.*
  E-mail: filipe.roseiro.cogo1@huawei.com

1. https://dependabot.com/
2. https://github.com/

tomate the process of updating and testing new releases from provider packages. Dependabot sits between a client's package manager and GitHub, observing all of 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. One 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 paper, we study the efficacy of Dependabot's strategy of leveraging the crowd to provide a compatibility score to help clients assess the risks involved 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 the range compatibility score dimension can help to 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 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 paper 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 risks involved 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[3] as a means to bootstrap other studies in the area.

The remainder of this paper is organized as follows. Section 2 introduces key concepts related to our study. Section 3 explains the employed data collection procedures. Section 4 presents the motivation, approach, and findings of our three research questions. Section 5 discusses the implications of our findings. Section 6 presents related work. Section 7 discusses the threats to the validity of our study. Finally, Section 8 concludes the paper.

3. https://www.dropbox.com/s/jv0i0s4xqb8e2ht/dependabot_compatibility_score_online_appendix.tar.gz

2 Background and Motivating Example

In this section, we present the key concepts related to automated dependency management with Dependabot (Section 2.1). We also present a motivating example (Section 2.2) to help illustrate the intentions of the compatibility score.

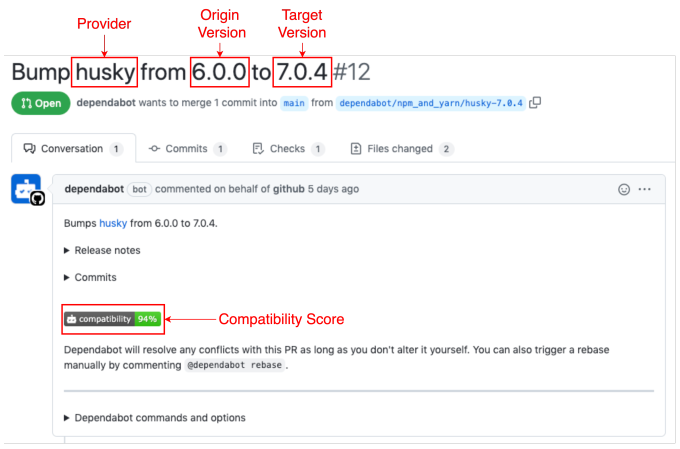
2.1 Dependabot

More and more packages are making use of tools that help to automate dependency management. For example, bots are increasingly being used to notify client packages in the form of PRs when one of their dependencies releases a new version. Clients can then configure a CI pipeline to automatically test the new dependency release in an attempt to verify that it is compatible with their package and does not contain any API-backwards incompatibility or introduce other regressions.

Dependabot is perhaps currently the most popular automated dependency-management tool, having first launched on May 26, 2017[4] and later being acquired by GitHub on May 23, 2019[5]. 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 of the providers a client package depends on. 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 of 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 dependency update (P, V_O, V_T). For each client with CI enabled (e.g., Travis CI[6] or GitHub Actions[7]) 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 1 provides an example of a Dependabot PR with the provider package name, origin version, target version, and the compatibility score highlighted.

4. https://dependabot.com/blog/introducing-dependabot/
5. https://dependabot.com/blog/hello-github/
6. https://travis-ci.com/
7. https://docs.github.com/en/actions



Bump husky from 6.0.0 to 7.0.4 #12

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

Conversation 1 Commits 1 Checks 1 Files changed 2

dependabot bot commented on behalf of github 5 days ago

Bumps husky from 6.0.0 to 7.0.4.

▶ Release notes

▶ Commits

compatibility 94%

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

Fig. 1: An example of a Dependabot PR with the provider package name, origin version, target version, and the compatibility score highlighted.

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[8]. Otherwise, the badge will simply say that the compatibility score is "unknown".

2.2 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"[9] (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

8. https://github.com/dependabot/dependabot-core/issues/4001#issuecomment-870399478
9. https://softwaretestingfundamentals.com/smoke-testing/

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 risks involved 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 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 or not 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.

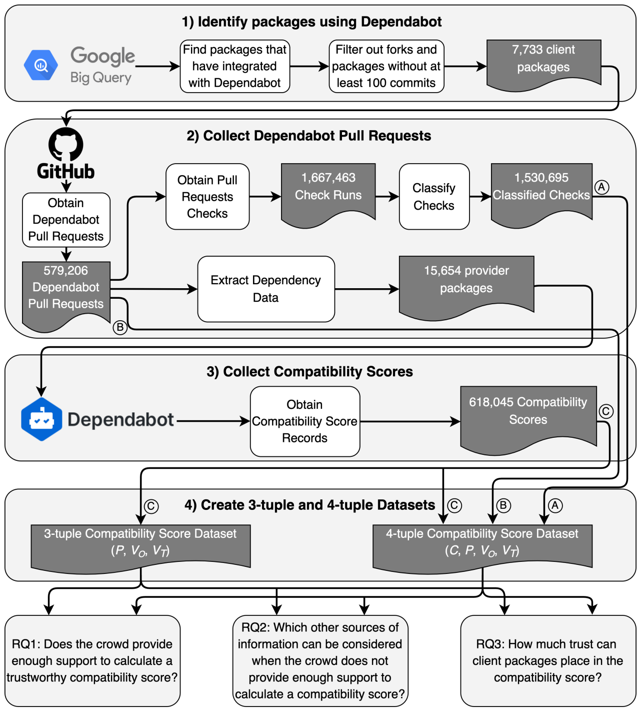
After a few days, Alice checks back on the PR, and finds that Dependabot reports that updating the dependency from the version that she currently uses to the newly released target version has a compatibility score of 100%. With this information in mind, she decides to merge the PR.

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.

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 2: (i) we identify packages on GitHub that use Dependabot, (ii) we collect all Dependabot PRs for each packages 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.



1) Identify packages using Dependabot
Google Big Query
Find packages that have integrated with Dependabot
Filter out forks and packages 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_O, V_T)
4-tuple Compatibility Score Dataset (C, P, V_O, V_T)
RQ1: Does the crowd provide enough support to calculate a trustworthy 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 packages place in the compatibility score?

Fig. 2: Overview of the data collection process.

3.1 Identify packages using Dependabot

To identify the packages using Dependabot, we leverage the Google BigQuery Public Datasets[10] 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 that GitHub contains some toy packages [10] 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 [3, 10, 11]. 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.

3.2 Collect Dependabot Pull Requests

We use the GitHub API[11] to retrieve all Dependabot PRs opened in the list of packages that we collected in Section 3.1. This step is necessary as a follow up to Section 3.1 as it allow us to extract the information about the provider

10. https://cloud.google.com/bigquery/public-data/
11. https://docs.github.com/en/rest

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 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 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.

To determine whether a client's CI pipeline passed or failed when testing each dependency update, we retrieve the GitHub Checks[12] that ran on each PR. Checks are pipeline runs or custom scripts that perform specific tasks (e.g., linters[13] 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. [6] when examining the percentage of open-source packages that have a CI pipeline configured.

Hejderup and Gousios [12] 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 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).

3.3 Collect Compatibility Scores

For each specific provider package we extracted in Section 3.2, we retrieve all compatibility scores using the Dependabot API[14]. 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.

12. https://docs.github.com/en/rest/reference/checks
13. https://github.com/collections/clean-code-linters
14. https://dependabot.com/compatibility-score/

3.4 Create 3-tuple and 4-tuple Datasets

Because our package list in Section 3.1 is a non-exhaustive set of package that use Dependabot, the Dependabot PRs we collect in Section 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 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 3.3 as the "3-tuple dataset".

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 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 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 the 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.

4 Findings

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

4.1 RQ1: 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 [13], particularly when the quality of test suites cannot be fully trusted – a relatively common scenario according to recent research [12]. 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[15] as well as developer blog sites[16]. Therefore, in this research question, we look to answer 1) how often do compatibility scores have the minimum number of candidate updates required to be shown as a badge on Dependabot PRs? and 2) 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 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 1)** ***The majority (83%) of dependency updates do not have enough candidate updates to display a compatibility score badge on Dependabot PRs.*** Figure 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 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 approximately two-fifths (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.

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

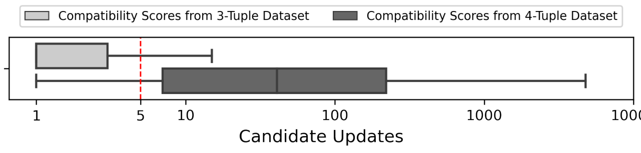


Fig. 3: The distribution of candidate updates for compatibility scores with at least 1 candidate update from the 3-tuple and 4-tuple datasets.

Observation 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 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 mislead 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.

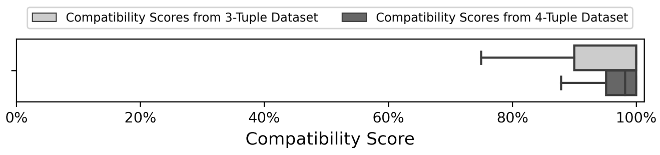


Fig. 4: The distribution of compatibility scores that have at least 5 candidate updates from the 3-tuple and 4-tuple datasets.

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.

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 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.

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[17], 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[18] [14, 15]. Similarly, we calculate three origin version range compatibility score metrics: a *patch origin verion 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 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 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.

17. https://semver.org
18. https://nodesource.com/blog/semver-tilde-and-caret/

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 risk involved 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 [12], which would bias how we assess the compatibility of the dependency update (e.g., low-quality CI pipelines might fail often 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.

TABLE 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 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 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 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.

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`[19] package in `R` as our random forest implementation due to its enhanced performance.

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 [16, 17] 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 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 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 or not 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 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

19. https://cran.r-project.org/web/packages/ranger/ranger.pdf

candidate updates as what is used by the raw compatibility score. Figure 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.

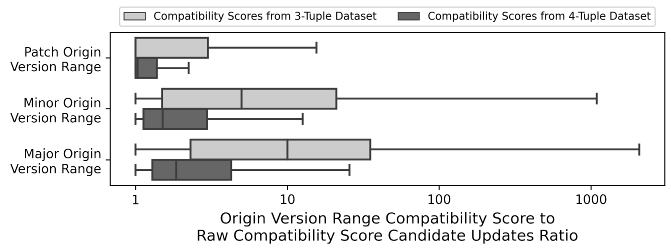


Fig. 5: The distribution of ratios of candidate updates for each origin version range compatibility score to the associated raw compatibility score from the 3-tuple and 4-tuple datasets.

Observation 5) ***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 6 shows the distribution of AUC improvements of both of these models compared to the baseline model median AUC. Figure 7a and Figure 7b 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.

Observation 6) ***Combining metrics from both dimensions into a single model results in a larger predictive power than each of the studied dimensions individually.*** Figure 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 7c shows the distribution of permutation importances of each metric for the model with combined metrics from both dimensions.

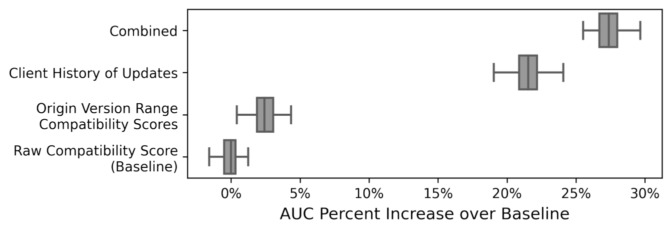


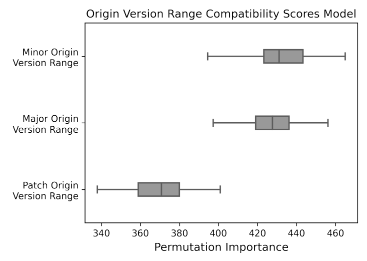
Fig. 6: The distribution of the improvement in AUCs of models constructed with an individually studied dimension, and with all studied dimensions combined, compared against the baseline model.

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

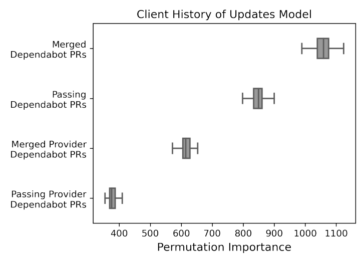
- Considering a range of origin versions to a specific target version helps to increase the number of candidate updates, effectively amplifying the knowledge from the crowd.
- Metrics from the origin version range compatibility scores and client history of updates dimensions can help improve the prediction 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.

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

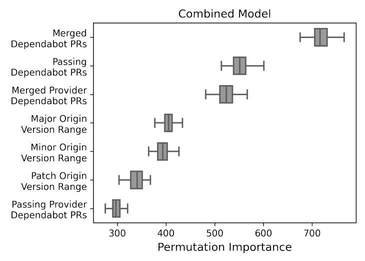
Motivation. As previously explained, the compatibility score for a 3-tuple dependency update is calculated as the ratio of successful updates to the total number of candidate updates. The problem here is that a compatibility score with 5 successful candidate updates (i.e., 100%) will result in the associated 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



(a) Distribution of permutation importance's of each metric from the origin version range compatibility scores model.



(b) Distribution of permutation importance's of each metric from the client history of updates model.



(c) Distribution of permutation importance's of each metric from the model with combined metrics from both dimensions.

Fig. 7: The distribution of permutation importance's of each metric for each model.

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 [18] (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) [19].

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 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 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 8.

Observation 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 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,

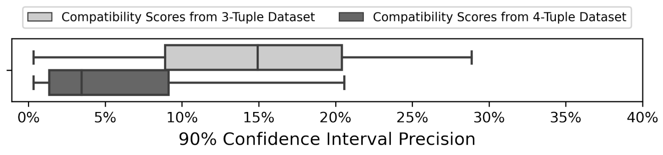


Fig. 8: The distributions of the 90% confidence interval precision for both the 3-tuple and 4-tuple datasets.

it is worth noting that it is common for client's tests to not thoroughly exercise the package's dependencies [12]. 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% 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 Discussion

In this section, we discuss the findings observed in Section 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 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 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 [15]. Assuming this policy is followed by provider package maintainers, which Decan and Mens [15] 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 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 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 [20, 21]. 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. [22] 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 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"[20], 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 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.

20. https://github.com/ossf/scorecard

6 Related Work

In this section, we discuss related work to automated dependency management (Section 6.1) and crowd-sourced software engineering (Section 6.2).

6.1 Automated Dependency Management

Multiple studies examine the growing trend of using third-party provider packages to build new software [23, 24], sometimes even for trivial tasks [1, 2, 25]. However, multiple studies [26–28] find that clients are reluctant to update their dependencies and that the difference between the client's currently used version of a provider and the provider's latest release (i.e., *technical lag*) is increasing over time. Existing research finds that this can lead to security issues [4, 29], depending on deprecated releases [30], and dependency conflicts [31].

Bogart et al. [13] find that the risk of breaking changes is one of the primary concerns that developers experience when it comes to dependency management, and multiple works study how to detect breaking changes in dependency updates [32, 33]. Most relevant to our paper, Hejderup and Gousios [12] find that client's tests can only detect on average 47% of direct and 35% of indirect artificial faults and note that a combination of static and dynamic analysis should be used to effectively detect breaking updates.

Automated dependency management bots have become a popular area of research [34–37]. Specifically, Mirhosseini and Parnin [3] find that clients who use dependency bots tend to upgrade their dependencies 1.6 times as often clients that do not. In the same vein, Alfadel et al. [11] find that 65% of Dependabot security PRs are merged and integrated in the packages, often within a day, and that the large majority of unmerged Dependabot security PRs (94%) were closed by Dependabot itself because the PR had become outdated.

6.2 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[21], a popular Question and Answer (Q&A) site, has been the topic of many studies [38–42] that explore how the job of answering questions related to software engineering is outsourced to the crowd. Specifically, Abdalkareem et al. [43] 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. [44] proposed an approach and associated tool to 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. [8] 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. [9] 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 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, are more likely to provide representative executions and only use the public parts of the provider package.

21. https://stackoverflow.com/

7 Threats to Validity

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

7.1 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 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 where 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 or not 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 or not 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.

7.2 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*[22] 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 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.

8 Conclusion

Today's software systems are rarely built from scratch, 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 risk involved 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 paper, we perform 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 risks involved 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 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.

9 Acknowledgment

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

References

[1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 385–395.

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

[3] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 84–94.

[4] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 181–191.

[5] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: cost negotiation and community

22. https://docs.renovatebot.com/merge-confidence/

values in three software ecosystems," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 109–120.

[6] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 426–437.

[7] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[8] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using Others' Tests to Identify Breaking Updates," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 466–476.

[9] G. Mezzetti, A. Møller, and M. T. Torp, "Type Regression Testing to Detect Breaking Changes in Node.js Libraries," in *Proceedings of the 32nd European Conference on Object-Oriented Programming*, 2018, p. 24 pages.

[10] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 92–101.

[11] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the Use of Dependabot Security Pull Requests," in *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories*, 2021, pp. 254–265.

[12] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? A case study of java projects," *Journal of Systems and Software*, p. 111097, 2021.

[13] C. Bogart, C. Kastner, and J. Herbsleb, "When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop*, 2015, pp. 86–89.

[14] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency Versioning in the Wild," in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories*, 2019, pp. 349–359.

[15] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[16] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An Empirical Comparison of Model Validation Techniques for Defect Prediction Models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

[17] D. Lee, G. K. Rajbahadur, D. Lin, M. Sayagh, C.-P. Bezemer, and A. E. Hassan, "An empirical study of the characteristics of popular Minecraft mods," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3396–3429, 2020.

[18] C. Davidson-Pilon, *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference*, 1st ed. Addison-Wesley Professional, 2015.

[19] A. Hazra, "Using the confidence interval confidently," *Journal of Thoracic Disease*, vol. 9, no. 10, pp. 4125–4130, 2017.

[20] D. A. Norman, *The design of everyday things*. Basic Books, 2002.

[21] T. Kulesza, S. Stumpf, M. Burnett, and I. Kwan, "Tell me more? the effects of mental model soundness on personalizing an intelligent agent," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012, pp. 1–10.

[22] Y. Zhang, Q. V. Liao, and R. K. E. Bellamy, "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*, 2020, pp. 295–305.

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

[24] A. M. Fard and A. Mesbah, "JavaScript: The (Un)Covered Parts," in *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 230–240.

[25] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adams, "On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[26] A. Decan, T. Mens, and E. Constantinou, "On the Evolution of Technical Lag in the npm Package Dependency Network," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 404–414.

[27] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is," in *Open Source Systems: Towards Robust Practices*. Springer International Publishing, 2017, vol. 496, pp. 182–192.

[28] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An Empirical Analysis of Technical Lag in npm Package Dependencies," in *New Opportunities for Software Reuse*. Springer International Publishing, 2018, vol. 10826, pp. 95–110.

[29] J. Cox, E. Bouwers, M. v. Eekelen, and J. Visser, "Measuring Dependency Freshness in Software Systems," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 109–118.

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

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

[32] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "APIDiff: Detecting API breaking changes," in *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 507–511.

[33] A. Møller and M. T. Torp, "Model-based testing of breaking changes in Node.js libraries," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations*

of Software Engineering, 2019, pp. 409–419.

[34] L. Erlenhov, F. Gomes de Oliveira Neto, R. Scandariato, and P. Leitner, "Current and Future Bots in Software Development," in *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering*, 2019, pp. 7–11.

[35] C. Lebeuf, A. Zagalsky, M. Foucault, and M.-A. Storey, "Defining and Classifying Software Bots: A Faceted Taxonomy," in *Proceedings of the IEEE/ACM 1st International Workshop on Bots in Software Engineering*, 2019, pp. 1–6.

[36] M. Wessel, "Enhancing developers' support on pull requests activities with software bots," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1674–1677.

[37] M. Wessel and I. Steinmacher, "The Inconvenient Side of Software Bots on Pull Requests," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 51–55.

[38] T. D. LaToza and A. van der Hoek, "Crowdsourcing in Software Engineering: Models, Motivations, and Challenges," *IEEE Software*, vol. 33, no. 1, pp. 74–80, 2016.

[39] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?: NIER track," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 804–807.

[40] C. Rosen and E. Shihab, "What are mobile developers asking about? A large scale study using stack overflow," *Empirical Software Engineering*, vol. 21, 2015.

[41] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? An analysis of topics and trends in Stack Overflow," *Empirical Software Engineering*, vol. 19, no. 3, pp. 619–654, 2014.

[42] B. Vasilescu, V. Filkov, and A. Serebrenik, "StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge," in *Proceedings of the International Conference on Social Computing*, 2013, pp. 188–195.

[43] R. Abdalkareem, E. Shihab, and J. Rilling, "What Do Developers Use the Crowd For? A Study Using Stack Overflow," *IEEE Software*, vol. 34, no. 2, pp. 53–60, 2017.

[44] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009, pp. 57–62.

[45] L. Hespanhol, C. S. Vallio, L. M. Costa, and B. T. Saragiotto, "Understanding and interpreting confidence and credible intervals around effect estimates," *Brazilian Journal of Physical Therapy*, vol. 23, no. 4, pp. 290–301, 2019.

[46] A. K. Gupta and S. Nadarajah, *Handbook of Beta Distribution and Its Applications*. CRC Press, Jun. 2004.

[47] A. K. Gupta, "Beta Distribution," in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 144–145.

Benjamin Rombaut is a MSc candidate with the Software Analysis & Intelligence Lab (SAIL) at Queen's University, Canada, as well as a Software Engineering Researcher at Huawei, Canada. He received his BSc in Software Engineering from the University of New Brunswick, Canada in 2019. His research interests include empirical software engineering, software supply chain management, and software bots. More information at: https://www.benrombaut.ca.

Filipe R. Cogo is a Software Engineering Researcher at Huawei, Canada. His research focuses on empirical software engineering and mining software repositories. He received his BSc and MSc in Computer Science from Universidade Estadual de Maringá (UEM), Brazil, and his PhD from Queen's University, Canada.

Ahmed E. Hassan is an IEEE Fellow, an ACM SIGSOFT Influential Educator, an NSERC Steacie Fellow, the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves/d on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. Contact ahmed@cs.queensu.ca. More information at: http://sail.cs.queensu.ca

APPENDIX A
PATTERNS FOR CLASSIFYING CI BUILD TYPES

The following steps are used to classify the checks that ran 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 2.

TABLE 2: 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`

APPENDIX B
DISTINGUISHING BETWEEN THE 3-TUPLE AND 4-TUPLE DATASETS

The list of client packages (Section 3.1) that we collected Dependabot PRs for (Section 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 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 packages 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 packages built in Section 3.1 is by no means an exhaustive list of all packages 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 packages.

If client C is in our list of packages, 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 package at any one time[23]. 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.

23. https://docs.github.com/en/code-security/supply-chain-security/keeping-your-dependencies-updated-automatically/configuration-options-for-dependency-updates#open-pull-requests-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 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 3.1. Then, the Dependabot PR shown in Figure 1 is collected in Section 3.2, where we determine that Dependabot has opened a PR to update the Husky provider package. Next, in Section 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.

Appendix 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 [45]. 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 modeling the compatibility score [46]. 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 [47]. That is,

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

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 [19].

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

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

Figure 9 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 9 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.

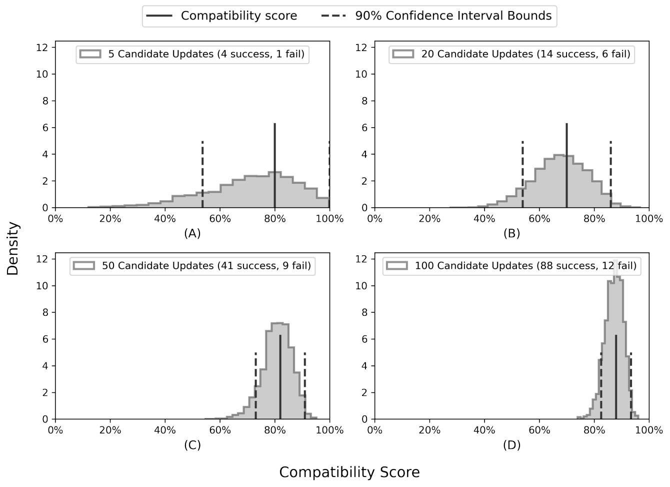


Fig. 9: Examples of posterior distributions for particular success/failure update pairs with relatively low (A and B) and high (C and D) candidate updates. The vertical lines mark the compatibility score (solid) and the upper and lower bounds of the associated 90% confidence interval (dashed).