

Dependency-Induced Waste in Continuous Integration



An Empirical Study of Unused Dependencies in the npm Ecosystem

NIMMI RASHINIKA WEERADDANA, University of Waterloo, Canada

MAHMOUD ALFADEL, University of Waterloo, Canada

SHANE MCINTOSH, University of Waterloo, Canada

Modern software systems are increasingly dependent upon code from external packages (i.e., dependencies). Building upon external packages allows software reuse to span across projects seamlessly. Package maintainers regularly release updated versions to provide new features, fix defects, and address security vulnerabilities. Due to the potential for regression, managing dependencies is not just a trivial matter of selecting the latest versions. Since it is perceived to be less risky to retain a dependency than remove it, as projects evolve, they tend to accrue dependencies, exacerbating the difficulty of dependency management. It is not uncommon for a considerable proportion of external packages to be unused by the projects that list them as a dependency. Although such unused dependencies are not required to build and run the project, updates to their dependency specifications will still trigger Continuous Integration (CI) builds. The CI builds that are initiated by updates to unused dependencies are fundamentally wasteful. Considering that CI build time is a finite resource that is directly associated with project development and service operational costs, understanding the consequences of unused dependencies within this CI context is of practical importance.

In this paper, we study the CI waste that is generated by updates to unused dependencies. We collect a dataset of 20,743 commits that are solely updating dependency specifications (i.e., the `package.json` file), spanning 1,487 projects that adopt npm for managing their dependencies. Our findings illustrate that 55.88% of the CI build time that is associated with dependency updates is only triggered by unused dependencies. At the project level, the median project spends 56.09% of its dependency-related CI build time on updates to unused dependencies. For projects that exceed the budget of free build minutes, we find that the median percentage of billable CI build time that is wasted due to unused-dependency commits is 85.50%. Moreover, we find that automated bots are the primary producers of dependency-induced CI waste, contributing 92.93% of the CI build time that is spent on unused dependencies. The popular Dependabot is responsible for updates to unused dependencies that account for 74.52% of that waste. To mitigate the impact of unused dependencies on CI resources, we introduce DEP-sCIMITAR , an approach to cut down wasted CI time by identifying and skipping CI builds that are triggered due to unused-dependency commits. A retrospective evaluation of the 20,743 studied commits shows that DEP-sCIMITAR  reduces wasted CI build time by 68.34% by skipping wasteful builds with a precision of 94%.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; **Empirical software validation**.

Additional Key Words and Phrases: continuous integration, unused dependencies, npm dependencies

ACM Reference Format:

Nimmi Rashinika Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the npm Ecosystem. *Proc. ACM Softw. Eng.* 1, FSE, Article 116 (July 2024), 23 pages. <https://doi.org/10.1145/3660823>

Authors' addresses: Nimmi Rashinika Weeraddana, University of Waterloo, Waterloo, Canada, nrweeraddana@uwaterloo.ca; Mahmoud Alfadel, University of Waterloo, Waterloo, Canada, malfadel@uwaterloo.ca; Shane McIntosh, University of Waterloo, Waterloo, Canada, shane.mcintosh@uwaterloo.ca.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2024 Copyright held by the owner/author(s).

ACM 2994-970X/2024/7-ART116

<https://doi.org/10.1145/3660823>

1 INTRODUCTION

Modern software systems are highly dependent upon ecosystems of reusable code [16, 62]. This reusable code takes the form of packages that are indexed by online package repositories, and are often orchestrated by package management systems, such as npm (JavaScript), PyPI (Python), and Maven (Java). While developing solutions that leverage software packages provides benefits, such as encouraging cross-project software reuse [8] and improving developer productivity [46], package-oriented reuse also introduces challenges for application developers [18, 46]. Indeed, prior work has shown that package-oriented reuse can create compatibility issues [10, 50, 74] and transitively propagate security vulnerabilities [4, 5, 74].

To mitigate such issues, maintainers actively update their packages, releasing new versions that fix bugs [15, 50, 74], enhance security [57, 58], improve performance [73], and introduce new features or other improvements [49, 73]. In fact, a median of 25% of npm packages are updated daily, while another 25% are updated every 5–22 days [18]. Given this rapid cadence of development, it is critical for developers who leverage these packages to actively maintain their dependency specifications to benefit from the latest updates.

As projects evolve, the management of dependencies tends to become more complex due to the continual incorporation of new dependencies and the obsolescence of existing ones. As such, projects have a tendency to accrue dependencies. This accrual frequently results in a considerable number of external packages remaining unused, despite being listed as dependencies [50, 63]. Identifying and removing such unused dependencies can be challenging for developers. Removing unused dependencies requires a careful round of time-consuming integration testing.¹ Given their largely benign perceived impact and the risky implications of erroneous removal [14], it is not surprising that prior work has found that it is not uncommon for rates of unused dependencies to reach 59% [50].

The accrual of unused dependencies is also associated with detrimental effects on Continuous Integration (CI) pipelines, i.e., the automatic build and test routines that are applied to the change sets that development teams produce [25]. In fact, unused dependencies bloat the CI build time of every build by spending additional time downloading and installing dependencies that are not strictly necessary. For example, in the discussion of the pull request #385 of the [E3SM-Project/e3sm_diags](https://github.com/E3SM-Project/e3sm_diags) project,² developers mention that the removal of unused dependencies would reduce the duration of CI builds for the project. Other discussions in GitHub projects also raised similar concerns about the accrual of unused dependencies and their negative impact on CI.^{3,4,5,6} To address such issues, prior work had suggested caching dependencies in the CI environment [28]. Also, CI providers, such as GitHub Actions (GHA),⁷ allow dependency caching to improve the overall CI build time.⁸

In addition to generating costs, updates to unused dependency versions cannot impact the build outcome as they do not affect build or runtime behaviour. Hence, all CI build time that is spent on change sets that solely update unused dependencies is wasted by consuming CI resources without providing value to development teams. This problem can be exacerbated by automated bots that update dependencies, potentially increasing the frequency of unnecessary updates [37]. Given that project budgets for CI are finite, such unnecessary resource consumption contributes to increased

¹<https://github.com/spacelabdev/spacelab-react/issues/335>

²https://github.com/E3SM-Project/e3sm_diags/pull/385

³<https://github.com/Jonbodo/Wildfire/pull/2>

⁴<https://github.com/ethereum/fe/pull/833>

⁵<https://github.com/rkorytkowski/openmrs-owa-conceptdictionary/pull/76>

⁶<https://github.com/ChristianCornelis/PersonalSite/pull/7>

⁷<https://docs.github.com/en/actions>

⁸<https://docs.github.com/en/actions/using-workflows/caching-dependencies-to-speed-up-workflows>

project costs. This overuse of resources has been shown to frustrate developers, illustrating the critical balance between resource allocation and effective project development [29, 51].

Therefore, the overarching goal of this paper is to *quantify*, *characterize*, and *mitigate* the CI waste that is generated by builds that are invoked due to updates to unused dependencies. As a concrete instance of this problem, we focus on npm dependencies in JavaScript projects that adopt GHA. To date, npm hosts more than four million packages and has the highest rate of growth in terms of packages among the most popular programming languages.⁹ We perform an empirical study on 20,743 commits and their corresponding CI builds that were triggered by updates to npm dependency specifications spanning 1,487 JavaScript projects. Our results are presented with respect to the following three dimensions:

Prevalence (Section 4). Understanding the degree to which waste of CI resources is generated by updates is crucial for CI consumers and providers. For consumers, such as developers and project maintainers, identifying substantial sources of waste can highlight the need for more efficient resource allocation or alternative optimization strategies. For CI service providers (e.g., GHA), recognizing this waste highlights opportunities for improving operational efficiency, whether through better resource allocation or optimizing CI processes. Our findings reveal that unused dependencies are a substantial source of waste in CI processes. From the perspective of the CI provider, we find that 55.88% (3,427 build hours) of the overall CI build time that is consumed by updates to npm dependency specifications in the studied projects is attributed to unused dependencies. At the project level, a median of 56.09% of CI build time is spent on updates to unused dependencies. To provide an operational cost perspective on this quantity of CI waste, we compare the waste of the most wasteful projects with the monthly budget of free build minutes that is provided to projects by GHA.¹⁰ Among those projects, we find that the CI build time that is spent on unused-dependency updates in 14 of the 54 studied months already exceeds their monthly allocation of free build minutes. A follow-up analysis (Section 7) shows that projects exceeding the budget of free build minutes waste a median of 85.50% of the total billable CI build time due to unused-dependency commits.

Source (Section 5). To tailor effective waste reduction strategies, we must better understand *who* is generating unused-dependency commits (i.e., bots or developers), and *which* types of dependencies tend to be affected (i.e., development or runtime). For example, if bots emerge as the primary contributors of unused-dependency commits, an effective waste mitigation strategy would need to include scrutinizing and refining the functionalities and configurations of these bots. Conversely, if developers are identified as the primary contributors of unused-dependency commits, an effective waste mitigation strategy would need to raise awareness about the importance of dependency management. Our analysis of the waste source reveals that a large proportion (92.93%) of the CI build time that is spent on unused dependencies is wasted due to bot-generated updates, with Dependabot¹¹ accounting for 74.52% of that wasted CI build time. With respect to the type of dependencies, the majority of the wasted CI build time (92.63%) occurs due to unused development dependencies, which are at lower risk of introducing field failures due to erroneous removal [19]. This suggests that development teams who are willing to invest in removing unused dependencies can focus on these development dependencies to reduce CI waste without exposing projects to elevated risk levels.

Mitigation (Section 6). Existing CI service providers are not inherently structured to mitigate the CI build time that is being consumed by unused-dependency commits, and developers are

⁹<https://libraries.io/NPM>

¹⁰<https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions>

¹¹<https://github.com/dependabot>

hesitant to remove unused dependencies in projects [49]. Thus, to cut down on this waste of CI resources, we introduce DEP-SCIMITAR_🐙—an approach that identifies commits that solely update unused npm dependencies and skip their associated builds. A retrospective analysis of the application of DEP-SCIMITAR_🐙 to past commits in the studied projects shows that 68.34% of wasted CI build time can be saved with a precision of 94%.

Contributions. In summary, this paper makes the following key contributions:

- We perform an empirical analysis of the phenomenon of dependency-induced waste in CI and its prevalence. Our study provides evidence of CI waste stemming from updates to unused dependencies, encompassing both the CI provider and consumer perspectives.
- We propose DEP-SCIMITAR_🐙—an approach to automatically detect and label unused-dependency commits that can be CI skipped.
- We develop a prototype implementation of DEP-SCIMITAR_🐙.^{12,13}
- We release a replication package that includes the dataset that we collected and the scripts that we implemented to conduct our analyses that span the three study dimensions.¹⁴

2 RELATED WORK

In this section, we situate our work with respect to the literature on dependency management (Section 2.1) and CI (Section 2.2).

2.1 Dependency Management

A plethora of prior work on dependency management has explored the impact of external dependencies on software development [7, 10, 12, 21]. For example, Basili et al. [7] studied the impact of dependency reuse on software quality and productivity. They concluded that the incorporation of external dependencies substantially benefits developers by mitigating defect density and reducing rework, ultimately enhancing productivity levels. Decan et al. [21] analyzed the growth of dependency networks over time. They observed a trend of consistent growth within these dependency networks in terms of size and the frequency of dependency updates.

Another popular line of work studied the consequences of relying on external dependencies. Several studies have discussed the increased exposure to security vulnerabilities that adopting dependencies incurs [3, 4, 11, 19, 23, 27, 53]. For example, Decan et al. [19] performed an empirical study of 399 security reports over a six-year period in the npm ecosystem. They found that the number of new vulnerabilities being identified and the number of affected dependencies grew over time. Bogart et al. [11] showed that ecosystems use different practices and policies to manage security vulnerabilities. Ferreira et al. [27] proposed an approach to mitigate supply chain attacks by enforcing a permission system for packages. The permission system only grants a package the minimal required permissions that it needs to function.

Other work studied the inefficiencies of dependencies that are not fully used [42, 54, 69]. For example, Wang et al. [69] performed a case study on the Chromium project,¹⁵ and found examples of dependencies that have low usage, e.g., at most 20% of the functionality that is provided by dependencies have been used by the studied projects. Jendele et al. [42] proposed a tool to decompose underused dependencies into a set of smaller components, which enables a finer granularity of reuse.

¹²<https://www.npmjs.com/package/dep-scimitar>

¹³<https://github.com/NimmiW/dep-scimitar>

¹⁴<https://zenodo.org/records/11192753>

¹⁵<https://www.chromium.org/chromium-projects>

Another line of research focused on bloated dependencies, i.e., dependencies that are bundled with additional source code that is never used to build or run the applications that depend on them [33, 59, 63, 64]. For example, Soto et al. [64] analyzed 9,639 Java artifacts that include 723,444 dependency relationships. This analysis revealed that 2.7% of the directly declared dependencies were bloated, 15.4% of the dependencies inherited from other sources were bloated, and 57% of the transitive dependencies in the studied artifacts were bloated. In another study, Soto et al. [63] analyzed dependencies of 435 Java projects, and found that bloated dependencies steadily increase over time, and 89.2% of the direct dependencies that are bloated remain bloated in all subsequent versions of the studied projects.

Recent studies quantify and characterize unused dependencies in projects [37, 41, 48, 50]. For example, Jafari et al. [41] analyzed 1,146 JavaScript projects to understand dependency smells. Their results revealed that some runtime dependencies that linger in dependency specifications of projects are not used by the source code—a phenomenon that the authors described as a dependency smell. Furthermore, Latendresse et al. [50] performed an empirical study on 100 JavaScript projects that use npm packages. Their results show that 59% of the runtime dependencies were never used in projects. Hejderup and Gousios [37] inspected 22 pull requests that were produced by the `DEPENDABOT` tool. They found that three of those pull requests provided upgrades to unused dependencies.

While previous studies have delved into the implications of the accrual of (unused) external dependencies, to the best of our knowledge, the wasted CI resources that are generated by their (unnecessary) maintenance have yet to be explored. Inspired by these prior studies, we set out to bridge that gap by empirically studying the quantity of waste in a sample of projects from the npm ecosystem.

2.2 Continuous Integration: Challenges and Solutions

CI offers numerous advantages to the software teams that adopt it [9, 40, 67]. For example, Hilton et al. [40] explored the adoption and implications of CI across 34,544 open-source projects, revealing that 40% of these projects adopt CI. Surveys of developers within these projects indicated that early bug detection capabilities and protection against build breakages are key incentives for CI adoption. Additionally, Vasilescu et al. [67] studied a dataset of 246 open-source projects that use CI and found that CI elevates developer productivity, especially within projects that adopt the Travis CI service.

On the other hand, adopting CI is not without its challenges. Several studies have discussed the hurdles that developers face, including troubleshooting woes and a preference for simpler CI configurations [32, 38, 39, 71]. For example, Hilton et al. [38] surveyed 523 developers to discover the barriers that they face. The survey results showed that 50% of the respondents reported having problems troubleshooting CI builds, and 52% of them prefer simplified configuration options for CI tools. Furthermore, CI build durations, which tend to grow as projects age, have been identified as a source of inefficiency [28, 29, 31, 70]. For example, Ghaleb et al. [31] used logistic regression to model long build durations using a dataset of 104,442 builds across 67 open-source projects. They observed that builds that are retried multiple times are most likely to be associated with long build durations. They also found that CI timeouts are one of the consequences of having long build durations, which is also supported by several other studies [29, 32]. Gallaba et al. [29] analyzed a dataset of 23.2 million CI builds spanning 7,795 open-source projects. They discovered a total of 17,917 timeouts, and suggested that heuristics, such as the frequency of timeout builds in the recent past, could be used to predict such builds. Weeraddana et al. [70] extended the work by Gallaba et al. [29] to characterize timeout builds. They found that timeout builds are strongly associated with the project build-history features and timeout-tendency features.

The need to restart builds has also been highlighted as a substantial drain on CI build time [24, 51, 61]. For example, Maipradit et al. [51] analyzed 66,932 code reviews from the OpenStack community. They found that 55% of code reviews invoke the recheck command after a failing build, and the build outcomes changed in only 42% of those cases. Durieux et al. [24] analyzed a dataset of 3,286,773 builds, and found that 56,522 of those builds were restarted due to timeouts and flaky tests, concluding that the restarted builds have an impact on the development workflow by slowing down the process of merging pull requests. Shi et al. [61] proposed a tool (IFIXFLAKIES) to mitigate the inefficiency of build rerunning, specifically in the context of flaky tests. This tool automatically fixes order-dependent tests to mitigate flakiness.

Efforts have also been made to reduce CI build time by skipping unnecessary builds or steps within builds [1, 2, 28, 44, 45]. For example, Gallaba et al. [28] proposed a language-agnostic approach (KOTINOS) to infer data from which build acceleration decisions can be made. They found that at least 87.9% of the 14,364 studied CI build records contained at least one Kotinos acceleration in their production setting. Abdalkareem et al. [2] examined 1,813 commits where developers requested for CI builds to be skipped. This analysis revealed reasons for skipping CI builds (e.g., when change sets only modify non-compileable files, such as documentation). Based on those reasons, the study proposed a rule-based method (CI-SKIPPER) that automatically identifies commits that could be CI skipped. In another study, Abdalkareem et al. [1] trained a decision tree classifier to detect CI-skippable commits, i.e., commits in which the updates made do not affect the source code or the functionality of the project, rendering these updates unnecessary for inclusion in the project's build execution [2]. Jin and Servant [44] proposed PRECISEBUILDSKIP—an enhanced CI-Skipper that considers additional rules to maximize the rate of build failure observation while minimizing the cost of CI.

Other studies proposed methods to predict build outcomes not only to provide an option to skip the predicted-to-pass builds, but also to provide early feedback for developers on build failures before the actual build fails [13, 35, 43, 60, 72]. For example, Chen et al. [13] proposed an approach to predict CI build outcomes by using history-aware and change set features, such as the status of the previous build and the total number of previously failed builds. Their predictor outperformed the state-of-the-art approaches by 47.5% in the F1-score for failed builds. Jin and Servant [43] also proposed a technique (SMARTBUILDSKIP) that detects CI build failures by separating the first failure from the remaining sequence of failures. Hassan and Wang [35] leveraged the build history data in order to predict CI build failures in Ant, Maven, and Gradle build systems.

Our study complements prior work that focused on approaches to speed up the CI build process. We specifically focus on CI builds that are triggered by dependencies, highlighting the problem of wasted CI build time due to unused dependencies, and characterizing the sources of such CI waste. Our empirical analysis provides insights to improve the omission of unnecessary builds that are triggered by unused dependencies. We leverage these insights to propose DEP-SCIMITAR¹ to cut down on wasted CI resources from both the CI provider and CI consumer perspectives.

3 STUDY DESIGN

In this section, we describe our study design. Specifically, we present our approaches to Project Selection (PS) and Data Curation (DC). Figure 1 provides an overview of the steps involved in these approaches, which we describe below.

3.1 Project Selection

Our study aims to analyze the waste in the CI process that is generated by version updates to unused dependencies. Therefore, we need to collect a dataset of projects that have accrued a rich

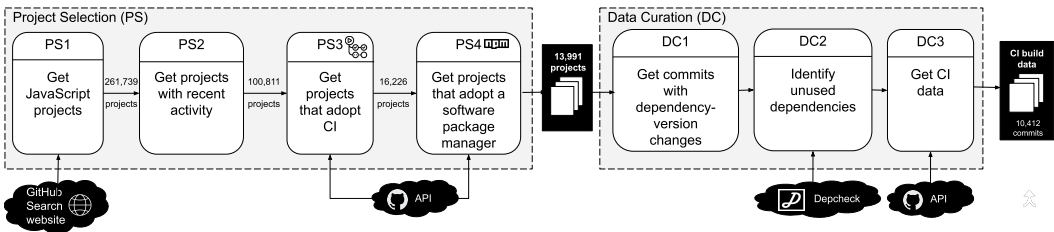


Fig. 1. An overview of our study design showing Project Selection (PS) and Data Curation (DC) steps.

history of dependency changes and build logs, as well as having a transparent CI configuration. Below, we describe the steps that we follow to select our sample of projects for analysis.

(PS1) Select JavaScript projects. We use the SEART GitHub search engine [17] to query for GitHub projects that meet our basic inclusion criteria. We select JavaScript projects due to JavaScript’s popularity and importance. In fact, JavaScript is currently the most popular programming language in the world, with a vibrant and fast-growing ecosystem.^{16,17} This rich ecosystem is a boon for developers, and our query returns 261,739 JavaScript projects.

(PS2) Select projects with recent activity. Since GitHub hosts toy and immature projects [17, 47, 55], we remove projects with fewer than ten commits. Inspired by prior work (e.g., [52]), we purposely do not restrict our project dataset only to the most active projects (i.e., projects with a large commit history) because many projects may not be updated frequently, but still play a critical role in the build process of other projects. Nonetheless, to ensure that the projects that we study have been active recently, we select projects that have received commits within the January 2020 to December 2022 timeframe. This filtering criterion improves the validity and modern relevance of the conclusions that we draw. After applying these filters, 100,811 projects survive.

(PS3) Select projects that adopt CI. To analyze CI waste, we need to select projects that actively apply CI. To do so, we choose to select projects that adopt the GHA CI service,⁷ which has quickly become the predominant CI service among npm projects on GitHub [34]. Furthermore, as of December 2022, GHA had accumulated a catalog of over 16,000 reusable Actions [20]. To identify projects that are configured for GHA, we use the GitHub API¹⁸ to check for the availability of the corresponding CI configuration files in each project. In particular, we check for the presence of `.yaml` files within the `.github/workflows` directories of the candidate projects.⁷ We find that GHA is configured for 16,226 projects in our dataset.

(PS4) Select projects that adopt a software package manager. npm is the de facto package manager used by JavaScript projects to manage their dependencies [68]. Therefore, we select projects that adopt npm. JavaScript projects that adopt npm must specify their dependencies in a `package.json` file, which lists the packages upon which this project depends, as well as their versioning constraints. A project with npm dependencies must contain at least one `package.json` file located in its root folder. To identify projects that use npm, we first clone a local copy of each of the 16,226 candidate projects that have been selected so far. Then, we search the root directory of the HEAD commit of each cloned repository for a `package.json` file. If a match is found, we store the repository for further analysis.

¹⁶<https://www.npmjs.com/>

¹⁷<https://libraries.io/npm>

¹⁸<https://docs.github.com/en/rest/repos/contents?apiVersion=2022-11-28>

At the end of this filtering process, 13,991 projects survive. These projects have a median of 181 commits and seven contributors. Our corpus of candidate projects comprises popular and large projects from organizations of influence, such as Meta,¹⁹ Google,²⁰ and Microsoft.²¹

3.2 Data Curation

After obtaining our set of projects, we process each project further to calculate the time that was spent on builds that were invoked due to updates to unused dependencies. Below, we describe the steps of this process in detail.

(DC1) Extract commits with dependency-version updates. We need to extract the dependency changes to identify updates to dependency versions. As explained above, JavaScript projects specify their dependencies in the `package.json` file, which contains the list of packages upon which the project depends. Hence, we extract all changes (i.e., commits) that modify the `package.json` file. Specifically, for each project, we mine through its commits, extracting the list of modified files, and the content of the modified lines using the `git-log` command. Note that to ensure the modern relevance of our analysis, we only select commits that occurred between 2020 and 2022 (inclusive). Then, we categorize these commits into those pertaining to dependency-version updates and those unrelated to dependency versions. Specifically, we consider a commit as a *dependency commit* if it exclusively modifies the `package.json` file²² and the only updates in that file are version specifiers of dependencies. Commits that do not meet both criteria are considered outside the scope of our investigation and are not considered in our analysis of dependency-induced CI waste. We detect 121,453 dependency commits spanning 1,854 projects.

Note that our approach may lead to the exclusion of commits that update unused dependencies in the `package.json` file while simultaneously making changes to other files that do not impact the source code functionality. For instance, commits that merely add comments in source files or modify the `README.md` file ([2]), in addition to updating unused dependencies in the `package.json` file, fall outside the scope of our analysis. As a result, the waste that we report represents a conservative lower bound, underestimating the actual quantity of dependency-induced CI waste.

(DC2) Identify unused dependencies. Following prior studies [41, 46, 56], we apply `DEPCHECK`²³ to identify unused dependencies that are listed in the `package.json` file. First, we run `DEPCHECK` on each dependency commit and store a list of all unused dependencies. Subsequently, we cross-reference this list of unused dependencies with those that are modified in the commit to identify relevant commits. When a match occurs, we classify the commit as a dependency commit responsible for modifying the version of an unused dependency, a.k.a., an *unused-dependency commit*. Of the 121,453 dependency commits that we extract in DC1, 49,731 are unused-dependency commits, which are of particular interest to us due to their potential to contribute to CI waste by triggering CI builds.

(DC3) Extract CI data. To analyze the amount of CI waste that is generated by unused-dependency commits, we retrieve the CI data that is associated with them using the GitHub API.²⁴ Note that not all of these commits are directly associated with CI builds. Some of them fail to trigger CI builds altogether, while in other cases, gathering CI data from the API is no longer possible, often because

¹⁹<https://github.com/facebookincubator/rapid>

²⁰<https://github.com/googlechromelabs/tooling.report>

²¹<https://github.com/microsoft/react-native-macos>

²²Note that we consider both the `package.json` file and the `package-lock.json` file

²³<https://github.com/depcheck/depcheck>

²⁴<https://docs.github.com/en/rest/checks/runs?apiVersion=2022-11-28>

the data is no longer available.²⁵ Consequently, we could retrieve CI data for only 20.9% of the unused-dependency commits.

We acknowledge that CI data is noisy [26, 30, 32]. This noise stems from various sources, such as experimental builds that fail, passing builds with ignored failing steps, and timeouts without proper signals. However, within the scope of our research, this noise does not pose a substantial concern because our study takes a holistic view of CI resources. In particular, we provide an estimation of CI waste from both the CI consumer and provider perspectives, which is a lower-bound estimate of the actual amount of CI waste. Indeed, since these noisy builds still consume CI resources, we consider them valid data entries for our study, and do not make attempts to filter them out.

Also, prior work [34] revealed that a number of GHA workflows are not entirely CI-related. For example, GitHub Actions are used for various purposes, such as manually triggering workflows (e.g., `workflow_dispatch`, accounting for 8.3%) and scheduled workflows (e.g., `schedule`, accounting for 8.1%). Our dataset does not contain such GitHub Actions that are not CI-related because we only collect the CI builds that are associated with dependency-update commits.

4 PREVALENCE OF CI WASTE DUE TO UPDATES TO UNUSED DEPENDENCIES

Measuring the time spent on CI builds is crucial for better resource management. By analyzing the effect of unused-dependency commits on CI resources, we strive to provide insights for the following two primary stakeholders of CI:

CI Consumers (i.e., project maintainers and developers). Understanding whether a considerable amount of CI build time is spent on unused-dependency commits will help CI consumers direct their future efforts. For example, if a large amount of CI build time is spent on unused-dependency commits, it may impose a financial burden on project maintainers. While some CI build time is provided for free on a monthly basis,¹⁰ CI build time that is spent on unused-dependency commits wastes this limited resource and may push projects over the free limit into billable time. If only a small amount of CI build time is spent on unused-dependency commits, it may suggest that effort would be better spent on other resource-saving options (e.g., build-oriented refactoring [65]).

CI Providers (e.g., GHA). If the CI build time that is spent on unused-dependency commits is not billable, the CI provider must absorb the cost of that CI build time. Such wasted resources that are spent across a large number of projects will quickly accrue. Even if the wasted CI build time is billable and the cost is borne by consumers, these wasted resources still indicate inefficiencies in resource allocation and present an opportunity for the optimization of CI operations.

4.1 Approach

Our quantification of CI waste resulting from version updates to unused dependencies offers insights that are tailored to the distinct perspectives of each CI stakeholder.

CI Consumer. In this perspective, we stratify our analysis based on individual projects. For each project in our dataset, we count the commits and compute the CI build time that is associated with builds that were triggered due to unused dependencies. This approach sheds light on the concern from the perspective of project maintainers, offering insights into how this problem impacts different projects within the open-source community. For example, it provides insights regarding the unnecessary maintenance activity that is generated by unused dependencies.

CI Provider. From this perspective, our investigation focuses on quantifying the combined influence of unused-dependency commits on CI build time. In other words, we count the commits and compute the CI build time that is associated with builds that were triggered due to unused dependencies. This viewpoint emphasizes the cost incurred by the CI provider.

²⁵GitHub API allows us to query data of CI builds for a limited number of the most recent days.

Table 1. The prevalence of CI waste from unused dependencies. The table presents the total and wasted number of commits and builds. The table further presents figures for both CI providers and consumers.

	CI Provider		CI Consumer	
	Commits	Build Hours	Commits (median)	Build Minutes (median)
Total Quantity (#)	20,743	6,133	7	42.13
Wasted Quantity (#)	10,412	3,427	3	23.63
Wasted Quantity (%)	50.19%	55.88%	42.85%	56.09%

4.2 Results

Table 1 shows the proportion of commits and build hours that are generated by version updates to unused dependencies from the CI provider and consumer perspectives.

Observation 1: *More than half (55.88%) of the CI build time for dependencies is spent on CI builds that are triggered by unused-dependency commits.* This equates to a substantial waste of 3,427 build hours, originating from 50.19% (10,412 commits) of all dependency commits. A closer inspection reveals that 30% (3,123) of these unused-dependency commits consume more than ten minutes of CI build time for each commit, and 7% (728) consume more than half an hour for each commit. For CI providers, the fact that unused-dependency commits make up over half of all dependency commits presents an opportunity to reduce waste.

Dependency update commits account for a median of 2.30% of the total number of commits during the studied period, whereas unused-dependency commits account for a median of 1.09%. Across all projects in our dataset, the overall percentage of dependency-update commits is 3.21%, and the percentage of unused-dependency commits is 1.60%. Despite these modest percentages, the impact on CI build time is non-negligible and should not be ignored.

Observation 2: *At the project level, a median of 56.09% of the CI build time that is spent on dependency commits is generated by unused-dependency commits.* The median CI build time that is consumed by unused-dependency commits in a project is 23.63 minutes. In fact, 30% of the projects in our dataset exceed an hour of wasted CI time. For example, the [dekkerglen/cubecobra](#) project consumed an hour of CI build time on unused-dependency commits, making up 31% of its total CI build time for all of its dependency commits. In more extreme cases, such as the [bus-stop/x-terminal](#) project, an alarming 127 build hours are wasted, with over 70% of its dependency-related CI build time being wasted on unused dependencies.

To provide an operational cost perspective on this quantity of CI waste, we compare the project-level waste with the budget of 2,000 build minutes per month that is provided to projects by the free plan of GHA.¹⁰ We conduct a focused analysis of the top six projects that accrue the largest amounts of CI waste. For this analysis, we use GHA's billing criteria. In fact, GHA calculates build minute usage for billing based on factors, such as the platform that was used (Linux, Windows, or MacOS) for the execution of the build. Thus, for each build (triggered by unused-dependency commits), we identify the platform on which the build was executed from the CI data that we retrieve from the GitHub API during the DC3 step in Section 3.

Figure 2 shows the number of build minutes that are consumed by unused-dependency commits per month. The figure shows instances where this wasted CI build time alone already exceeds the entire monthly budget of free CI build time for these projects. Indeed, 14 of the 54 studied monthly periods exceed the free monthly budget. In the most extreme case, the [bus-stop/x-terminal](#) project wasted 9,756 build minutes in April 2022, exceeding the entire monthly budget of free build minutes by almost fivefold. In other months, the wasted CI build time of this project still constitutes

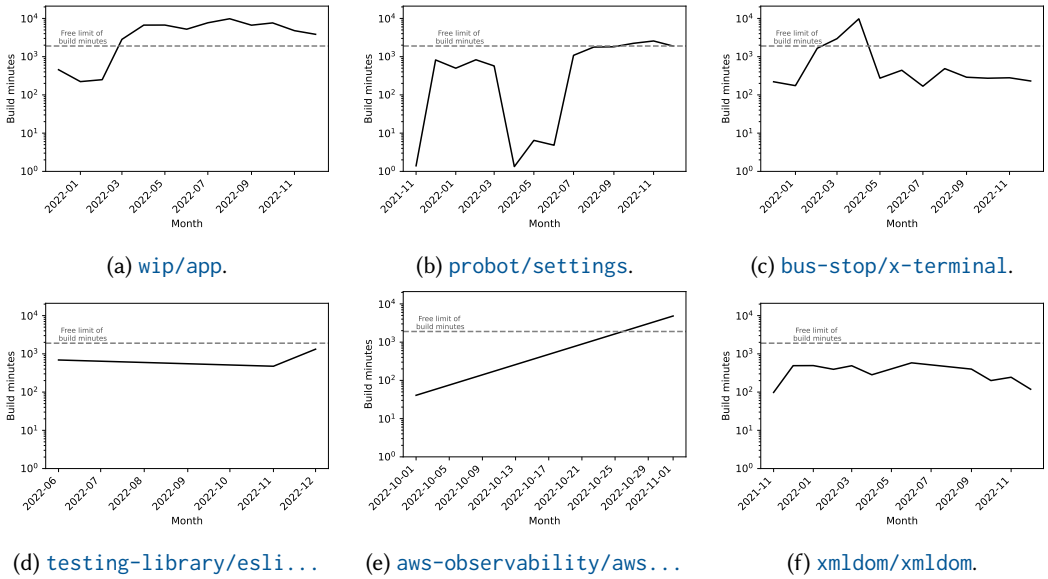


Fig. 2. CI build time consumption of top six projects (`wip/app`, `probot/settings`, `bus-stop/x-terminal`, `testing-library/eslint-plugin-jest-dom`, `aws-observability/aws-otel-js`, and `xmldom/xmldom`) per month due to unused-dependency commits. The graphs corresponding to other projects are provided in our online appendix.¹⁴

a substantial portion, comprising at least 8.4% (168 build minutes) of the available free CI build time for the project.

Although both successful and failed builds triggered by updates to unused dependencies are considered wasteful due to their resource consumption, one may argue that failed builds are not actually wasteful because they usually raise concerns that developers should address. To address this viewpoint, we conduct a follow-up analysis to examine the percentage of successful and failing builds in our dataset. We find that 87.61% of the builds that are associated with unused-dependency updates are successful, while 12.39% failed. To explore the impact of considering only successful builds as wasteful, we conduct a revised prevalence analysis. This analysis reveals that wasted builds from unused-dependency commits account for 43.98% of the dependency-update commits in our dataset, with wasted CI build time comprising 38.19%.

Unused dependencies are a substantial source of inefficiency in CI processes. For CI providers, more than half (55.88%) of the CI build time of dependency-update commits is taken up by unused-dependency commits in the studied projects, generating a considerable amount of waste (3,427 build hours). At the project level, the median project spends 56.09% of its dependency-related CI build time on updates to unused dependencies. Among the six most wasteful projects, more than their entire monthly budget of 2,000 free build minutes is entirely spent on building unused-dependency commits in 14 of the 54 studied months.

5 SOURCE OF CI WASTE DUE TO UPDATES TO UNUSED DEPENDENCIES

In Section 4, we observe that a considerable amount of CI waste is generated by unused-dependency commits. Understanding the origin of such commits is essential for crafting targeted solutions. In this section, we characterize CI waste according to the type of (1) commit author and (2) dependencies being updated.

Commit Authorship. Unused-dependency commits might be created by people maintaining the project or an automated software bot. Prior research [6, 36, 53] suggests that projects often use automated bots, such as Dependabot, to keep their dependencies up to date. If bots emerge as the primary contributors of unused-dependency commits, an effective waste mitigation strategy would need to include scrutinizing and refining the functionalities and configurations of these bots. Conversely, if developers are identified as the primary contributors of unused-dependency commits, an effective waste mitigation strategy would need to raise awareness about the importance of dependency management. The time that developers are spending on this unnecessary maintenance of unused dependencies could be better spent on more productive and impactful development activities.

Dependency Type. According to official guidelines,²⁶ npm dependencies may be *development* and *runtime*. Development dependencies are used during development and testing, and are listed in the `devDependencies` section of the dependency specification file (`package.json`). For instance, `webpack`²⁷ is a development dependency in JavaScript projects, bundling modules for delivery on the web. Runtime dependencies are necessary for production deployment environments. An example would be `lodash`,²⁸ which provides implementations of data structures like arrays and strings.

We strive to understand the type of dependencies that generate most of the waste to formulate effective mitigation strategies. For example, if most unused-dependency commits update development dependencies, they would present less risk-prone opportunities for optimization, as they are used solely during the development phase and are not installed in production environments.

5.1 Approach

Our approach to understanding the sources of unused-dependency commits focuses on (1) identifying who made the commit and (2) categorizing the type of dependency.

Commit Authorship. To distinguish between bot-generated and developer-generated waste, we identify who authored each unused-dependency commit. Following prior work [22], we apply a regular expression to detect authors having the term “bot” in their name, classifying them as bots. All other authors are labeled as developers. Our analysis reveals four bot candidates, and these are indeed automated bots: Dependabot,¹¹ Renovate bot,²⁹ Snyk bot,³⁰ and Depfu bot.³¹ For the remaining authors, we estimate the accuracy by inspecting a sample of 400 randomly selected commits, which provides a 95% confidence level that our observed proportions are within a confidence interval of $\pm 5\%$. We inspect each sampled commit to determine if its author is a bot. If inspection of the author’s name is inconclusive, we then cross-reference the name with the corresponding GitHub profile to arrive at a decision. This analysis yields no instances of incorrect labeling. To account for potential name variations or aliases, we employ heuristics to consolidate

²⁶<https://docs.npmjs.com/specifying-dependencies-and-devdependencies-in-a-package-json-file>

²⁷<https://webpack.js.org/concepts/>

²⁸<https://lodash.com/>

²⁹<https://www.mend.io/renovate/>

³⁰<https://snyk.io>

³¹<https://depfu.com>

Table 2. Distribution of unused-dependency commits and corresponding build hours over bots and developers.

	Bot	Developer
Commits	9,280 (89.12%)	1,132 (10.88%)
Build hours	3,184 (92.93%)	242 (07.07%)

Table 3. Build hours attributed to unused-dependency commits authored by bots.

Bot	Commits	Build Hours
Dependabot	6,541	2,373
Renovate bot	2,514	741
Snyk bot	174	60
Depfu bot	51	10
Total consumption	9,280	3,184

identities [66]. After establishing commit author categories, we compute both the number of unused-dependency commits and the total CI build time that is consumed by these commits.

Dependency Type. To understand how dependency-induced waste is associated with the different dependency types, we examine the nature of dependencies that cause CI waste. We group unused dependencies into development and runtime categories according to whether they appear in the `devDependencies` or `dependencies` section of the `package.json` file, respectively. We analyze each unused-dependency commit by extracting both development and runtime dependencies from the `package.json` file as of the commit's timestamp. After obtaining the list of dependencies, we cross-reference it with the unused dependencies that we identify for each commit in Section 3 (DC2) to determine whether an unused dependency is a development or runtime dependency. Then, we calculate the CI waste for each dependency type by counting the number of wasteful builds and their associated wasted build hours.

5.2 Results

Tables 2 and 3 present the number of commits and build hours that are associated with unused-dependency commits, respectively.

Observation 3: *Bots are the primary contributors to the wasted build hours.* Table 2 shows that bots are responsible for a substantial proportion of the CI waste being generated by updates to unused dependencies. This consumes 3,184 build hours (92.93%) spanning 9,280 bot-generated unused-dependency commits (89.12%). In contrast, developers account for only 7.07% of this wasted CI build time. The 1,132 developer-generated unused-dependency commits are produced by 265 developers. Even though this percentage is much lower than that of bots, it is important to highlight that a small number of developers are responsible for a substantial proportion of the generated waste. For example, the two developers who produced the most unused-dependency commits contributed 48% of the wasted CI build time. This suggests that misinformed developers can quickly accumulate waste due to unnecessary maintenance.

Table 3 breaks down how each of the four detected bots contributes to CI waste. Dependabot emerges as the top contributor, representing roughly three-quarters of the wasted build hours (74.52%). Renovate bot is next, accounting for 741 hours (23.27%). The remaining Snyk bot and Depfu bot contribute 60 hours (1.88%) and ten hours (0.31%), respectively. This distribution closely follows the quantiles of unused-dependency commits that are associated with each bot.

Observation 4: *Unused development dependencies lead to most of the wasted build hours.* In Table 4, we see a clear distinction between the impact of unused development and runtime dependencies.

Table 4. Comparison of the total number of commits and build hours stemming from unused-dependency commits between development and runtime dependencies across all the projects in our dataset.

	Unused development dependencies	Unused runtime dependencies
Commits	8,762 (84.15%)	1,650 (15.85%)
Build hours	3,174 (92.63%)	253 (07.37%)

Table 5. Build hours resulting from unused-dependency commits in development dependencies.

Unused development dependency	Build Hours
@vercel/node	797
eslint	325
prettier	245
jest	209
mocha	112
Other development dependencies (937 dependencies)	1,485
Total consumption	3,174

Unused development dependencies contribute 92.63% of the total wasted time, amounting to 3,174 build hours. In contrast, unused runtime dependencies contribute only 7.37% (253) of the build hours. This suggests that development teams that are inclined to allocate resources to the mitigation of CI waste that is generated by unused dependencies would benefit most from focusing on these development dependencies. By doing so, they can mitigate CI waste efficiently while maintaining a minimal risk level of field failures, since these development dependencies are unlikely to affect the production environments [19].

Upon analyzing individual projects, we find that version updates to unused development dependencies consume a median of roughly five minutes of CI build time, while unused runtime dependencies consume a median of roughly three minutes; however, the distribution is skewed. Indeed, 19% of projects consume more than an hour and 40 minutes of wasted CI build time from unused development dependencies. Comparatively, only 4% of projects waste that much CI build time for runtime dependencies.

Furthermore, Table 5 shows that while a total of 942 development dependencies are associated with unused-dependency commits, only five of them collectively contribute to 53.21% of the overall CI build time for unused development dependencies, which highlights the critical role being played by specific development dependencies that are implicated in unused-dependency commits.

Bots contribute the vast majority (92.93%) of the CI build time that is wasted on unused dependencies. Moreover, unused *development* dependencies represent 92.63% of the wasted CI build time that is associated with unused dependencies in the studied projects. Thus, we recommend that waste mitigation strategies (1) target the most wasteful bots (i.e., Dependabot and Renovate bot) for improvements and (2) focus waste reduction efforts on development rather than runtime dependencies.

6 MITIGATION OF CI WASTE DUE TO UPDATES TO UNUSED DEPENDENCIES

Our results show that a substantial quantity of CI waste is generated by unused-dependency commits. Existing CI service providers do not have measures in place to minimize this waste. Consumers of CI services might not realize that unused dependencies linger in their projects [50]. Removing unused dependencies can be an onerous task. For example, a discussion on the [GoogleCloudPlatform/](https://cloud.google.com/)

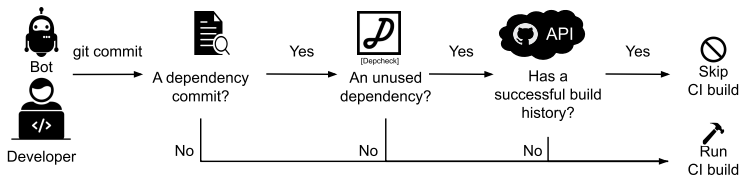


Fig. 3. An overview of the DEP-SCIMITAR workflow.

`nodejs-docs-samples` project suggests that refactoring code to remove such dependencies requires substantial effort.³² Moreover, Kula et al. [49] found that developers are hesitant to make dependency-related changes in general due to the substantial efforts needed to avoid introducing errors.

Given the hesitancy among developers to introduce changes into a stable codebase due to the perceived risks, opting to skip unnecessary builds when they are triggered by unused-dependency commits—which are skippable because they do not impact the project’s functionalities—emerges as a practical strategy. We perform an analysis of 272 GitHub issue reports related to removing unused dependencies (which is sufficient to provide a 95% confidence level with a 5% margin of error when making inferences about the entire population), and find that, in 40.44% of the cases, developers decide to remove the unused dependencies, while in 37.13% of the cases, developers decide not to remove unused dependencies; the other 22.43% of cases are not related to developer decisions. A more detailed description of this analysis is discussed in our online appendix.¹⁴

With these challenges in mind, we set out to develop an approach to mitigate CI waste by skipping builds when they are triggered by unused-dependency commits. This approach avoids the removal of such dependencies from the dependency specification files.

6.1 Approach



We propose DEP-SCIMITAR—an approach to cut down on dependency-induced CI waste. Our approach is agnostic of the CI service provider, and is freely available as an npm package to foster its adoption.¹² Figure 3 provides an overview of the design of DEP-SCIMITAR, emphasizing its automated reasoning process to skip CI builds for unused-dependency commits. Since unused-dependency commits are generated by humans as well as bots, DEP-SCIMITAR can process commits that are produced by both types of authors.



DEP-SCIMITAR begins when a commit generates a build request by analyzing the commit to check for updates to versions of dependencies in the `package.json` file. When such updates are detected, DEP-SCIMITAR uses the DEP-CHECK tool to detect whether the changed dependency has an unused status. Although tempting, simply skipping such commits without considering their recent history may inadvertently permit a failing build to pass. For example, commit `#fa843b6` of the `aws-observability/aws-otel-js` project is an unused-dependency commit and it is linked to a failing build. A closer inspection revealed that its parent commit is also linked to a failing build, and the builds of both the current and parent commits failed due to the same cause, i.e., a failure in the build step for testing a certain application feature. This implies that the error was carried forward without being fixed. To avoid such situations, whenever DEP-SCIMITAR detects an unused-dependency commit, it also retrieves the build status of the parent commit using GitHub API;²⁴ the tool skips the current build only if the build of the parent commit was successful. Note that our tool supports only public GitHub repositories due to its dependency on GitHub API calls

³²<https://github.com/GoogleCloudPlatform/nodejs-docs-samples/pull/3168>

for fetching build statuses without authentication tokens. For private projects, a project-specific authentication token is required.

6.2 Configuration


The DEP-SCIMITAR  prototype can seamlessly integrate into the CI process of consumers that adopt CI services, such as GHA,⁷ Travis CI,³³ and CircleCI.³⁴ Listing 1 shows a fragment from a GHA configuration file that uses DEP-SCIMITAR . The full version of this file is available in our online appendix.¹⁴ This file needs to contain steps to both install our tool (`npm install dep-scimitar`) and run it (`npm run dep-scimitar runremote`), as shown on lines 3 and 4 of Listing 1. Running the command classifies commits as unused-dependency commits or otherwise. This classification is stored as an environment variable (in line 5) and subsequently guides execution processes in the project, illustrated in line 8 (i.e., project-specific execution decisions).

DEP-SCIMITAR  offers another enhancement configuration to boost developer awareness. It can automatically add the `[CI skip]` tag to commit messages. This tag indicates to the CI service provider that the commit should not trigger a CI build. Commits with the `[CI skip]` tag skip builds on widely-used CI platforms, such as GHA, Travis CI, and CircleCI. Users can leverage this feature by installing DEP-SCIMITAR  locally and enabling it for their project using the command: `npm dep-scimitar on`.

```



1 - name: Check For Unused-Dependency Commits
2   run: |
3     npm install dep-scimitar # Install the tool
4     unusedDepChange=$(npm dep-scimitar runremote) # Run the tool
5     echo "UnusedDepCommit=$unusedDepCommit" >> $GITHUB_ENV # Env variable
6
7 - name: Project-specific Steps
8   if: ${{ env.UnusedDepCommit == 0 }} # Check the env variable
9   run: |
10    # project-specific steps

```

Listing 1. Example of a configuration file required for DEP-SCIMITAR .

6.3 Evaluation

To evaluate our approach, we create a prototype implementation that is compatible with npm-based GitHub projects. We conduct a retrospective analysis of past commits to the projects in our dataset, focusing on (1) the precision of the tool and (2) the reduction of wasted CI build time.

We find that the precision of DEP-SCIMITAR  is 94%. Having such high precision shows that the tool makes only a few mistakes where it falsely skipped builds that should not have been skipped. In particular, if the previous commit of an unused-dependency commit is linked to a passing build, DEP-SCIMITAR  skips the current commit because it updates an unused dependency and the commit does not have a failing history; yet, the current commit may fail if we do not skip it, and those cases are the false positives detected by our tool. A closer inspection reveals three scenarios that contribute to false positives, where the actual outcome of the build remains unknown. First, CI builds can fail due to external network issues, such as in the `mdn/bob` project, where a commit updating an unused dependency failed because it exceeded the API rate limit.³⁵ Second, CI build timeouts occur when the build process exceeds the allocated upper limit of build minutes, leading to

³³<https://www.travis-ci.com/>

³⁴<https://circleci.com/>

³⁵<https://github.com/mdn/bob/commit/013ff>

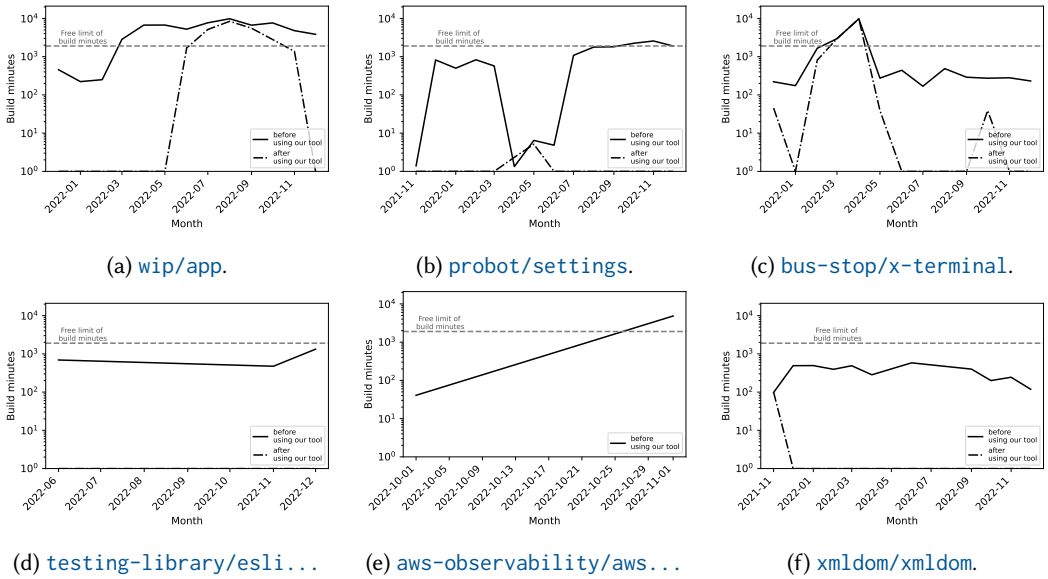


Fig. 4. Distribution of the wasted CI build time of the top six projects (`wip/app`, `probot/settings`, `bus-stop/x-terminal`, `testing-library/eslint-plugin-jest-dom`, `aws-observability/aws-otel-js`, and `xmldom/xmldom`) before and after applying our tool.

automatic cancellation; for instance, a commit³⁶ in the `btargac/excel-parser-processor` project that updated an unused dependency caused the build to time out and be marked as a failure. Third, premature cancellations of CI builds will also result in a “Failed” status. For example, a commit³⁷ in the `optimistiksas/oibus` project is associated with a failing build due to three canceled steps, despite its parent commit being linked to a successful build.

Note that calculating the recall is challenging. Below, we discuss two potential scenarios where DEP-SCIMITAR may produce false negatives. First, we only choose to skip a build that is triggered by an unused-dependency commit if its preceding build succeeds. Upon closer examination, we discover instances of unused-dependency commits where the previous build fails. Such preceding failures occur due to premature termination by the CI provider, e.g., when a higher-priority build is waiting in the queue. If the CI provider had not interrupted this build, it might have been successful. Such false-negative cases occur when our tool does not skip commits with a recent history of build failures and/or when commits include both an unused dependency version update and other minor changes, such as edits to documentation or comments in source files [2].

At the project level, we find the median number of commits that are skippable is three, with a waste of 23 minutes; however, we find that 25% of the studied projects can save at least an hour and 23 minutes of CI build time. Projects like `probot/settings` skip as many as 43 commits while saving 246.85 build hours. For a more detailed analysis of how such extreme projects generate waste in the context of the monthly budget of free build minutes that are provided by GHA, in Figure 4, we plot the amount of wasted CI build time before and after applying DEP-SCIMITAR, for the six projects that accrue the most dependency-induced CI waste. Focusing on these projects allows our analysis to effectively showcase the substantial savings and efficiency gains that are achievable with

³⁶<https://github.com/btargac/excel-parser-processor/commit/5ba1>

³⁷<https://github.com/iobroker/iobroker.pushover/commit/905a>

the DEP-SCIMITAR approach. This selection also acts as a proof of concept, indicating that similar benefits, if not proportionally scaled, could be achieved for other projects. For interested readers, the graphs that correspond to other projects are provided in our online appendix.¹⁴ Time (in terms of months) is shown on the x-axis, and the number of build minutes that are spent is shown on the y-axis. The solid line shows the CI build time that was spent for all unused-dependency commits before applying the tool, and the dashed line shows the CI build time that would be spent after applying the tool. The gap between the two lines demonstrates the CI build time that is safely skippable by the tool.

Figure 4 shows that for the `probot/settings`, `xml/dom/xml/dom`, and `testing-library/eslint-plugin-jest-dom` projects, the CI waste is cut down to almost zero. For the `wip/app` and `bus-stop/x-terminal` projects, we observe that there are months when DEP-SCIMITAR cannot skip many builds. This tends to occur when builds are linked to unused-dependency commits that were preceded by build failures. Nonetheless, we observe that a substantial amount of CI build time could still be saved for those projects in other months. Note that the `aws-observability/aws-otel-js` project has only two builds that are triggered by unused-dependency commits, and their parent commits are linked to failing builds, which DEP-SCIMITAR conservatively chooses to run. On the other hand, from the CI provider perspective, our tool detects 83% of wasteful CI builds, and could have saved 2,342 (68.34%) build hours.

Our DEP-SCIMITAR approach can effectively identify and skip CI builds that are associated with unused-dependency commits with a precision of 94%. A retrospective evaluation shows that integrating DEP-SCIMITAR in the studied projects would have saved 2,342 build hours, which amounts to 68.34% of the overall dependency-induced CI waste.

7 EVALUATION OF CI WASTE UNDER BILLING CONSTRAINTS

To analyze the sensitivity of our results for projects that pay for builds, we conduct a new analysis focusing on monthly periods during which projects exceed the free minute quota of 2,000 minutes.¹⁴

7.1 Approach

We calculate the billable CI build time for unused-dependency commits by summing up the build minutes that were spent on unused dependency updates. We select projects that exceed the 2,000-minute free quota due to these updates for further analysis for each monthly period of each studied project. By solely considering the build minutes that were spent on dependency updates, we establish a strict lower limit on instances where projects exceed this threshold, underestimating both the total number of projects exceeding the monthly limit and the overall billable CI build time that was wasted.

To illustrate our approach, we provide a set of examples. In April 2022, the `bus-stop/x-terminal` project spent 13,510 build minutes on all dependency-update commits, with 9,756 of those minutes being allocated to updates for unused dependencies. Consequently, 11,510 billable build minutes accrued (i.e., 13,510 – 2,000 free minutes). Of those billable minutes, 9,756 were spent (wasted) on updating unused dependencies. Similarly, in the `wip/app` project (in April 2022), 7,727 build minutes were spent on all dependency-update commits, with 6,689 minutes being associated with updates to unused dependencies. Hence, 5,727 billable build minutes accrued. In this case, the number of build minutes that were spent on updating unused dependencies (6,689 minutes) exceeded the number of billable build minutes (5,727 minutes), suggesting that the entire billable CI build time could have been saved if the build activity that was induced by unused dependencies had been avoided.

Similarly, we calculate the wasted billable minutes on unused dependency updates for all projects with billable minutes. Then, we aggregate these figures across the projects. Lastly, we compute how many billable minutes DEP-SCIMITAR⁴ could have saved per project through a replay analysis, identifying skippable build minutes that were induced by unused-dependency commits.

7.2 Prevalence of CI Waste

This analysis reveals that the median project incurs 1,831 billable minutes per month, with 1,446 of these minutes being wasted due to unused-dependency commits. For the studied dependency-update commits, 91,541 of the 107,067 total billable minutes (85.50%) were wasted due to unused-dependency commits. This rate of wasted billable build minutes is substantially higher than the overall percentage of wasted build minutes that we report in Section 4 (regardless of whether it is billable or not), i.e., 56.09% across all projects in Table 1.

7.3 Mitigation of CI Waste

The replay analysis reveals that DEP-SCIMITAR⁴ can safely skip 71,409 billable build minutes—a reduction of 66.69% (i.e., $\frac{71,409}{107,067} \times 100\%$). This aligns with our earlier finding in Section 6 where we report that DEP-SCIMITAR⁴ yields a 68.34% reduction in the total number of wasted build minutes due to unused-dependency commits regardless of being billed or not.

8 THREATS TO VALIDITY

Threats to Internal Validity. Inspired by prior work [22], we apply a regular expression to distinguish between commit authors who are bots and developers. Through this process, we identify four bots that are responsible for 9,280 unused-dependency commits. To mitigate false positives in our classification, we inspect the profiles and commit activity of these four bots and a sample of 400 commits; however, it is unlikely that our set of bots is complete. Thus, our bot-produced unused-dependency commit rates are best seen as minimum estimates rather than exact figures. In a similar vein, since GitHub allows the same commit author to use different names and/or different email addresses when committing changes,³⁸ there is likely noise in our authorship analyses (Section 5). To reduce the impact of aliases, we apply heuristics to consolidate them [66]; nonetheless, our findings should be viewed as estimates rather than precise figures.

Threats to Construct Validity. We use the DepCheck²³ tool to detect unused dependencies. Hence, we are limited by the accuracy of this tool. Validating unused dependencies can be challenging since npm packages can be loaded dynamically (e.g., using reflection mechanisms) that can go undetected by a purely static analysis. Despite the potential for false positives, DepCheck is, to the best of our knowledge, the de facto standard tool for detecting unused JavaScript dependencies, being relied upon by several prior work [41, 46, 56]. Furthermore, the tool actively addresses false positives,³⁹ making it a practical choice for both developers and researchers.

Threats to External Validity. Our study is based on JavaScript projects that use npm and adopt GHA from 2020 to 2022 (inclusive). Hence, our findings might not directly apply to projects developed in other programming languages, those that use other CI providers, or commits made outside the specified timeframe. Also, our study's reliance on the free build minutes quota as of August 2023 introduces the possibility that our findings may become outdated if there are changes to GitHub's billing structure.¹⁰ Despite these specifics, the key concept and the design of our study can still be applied to other settings to expand the investigation of dependency-induced CI waste.

³⁸<https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-email-preferences/setting-your-commit-email-address>

³⁹<https://github.com/depcheck/depcheck/releases>

9 CONCLUSION AND LESSONS LEARNED

In this paper, we study dependency-induced CI waste. We collect and analyze a dataset of 20,743 dependency commits from 1,487 JavaScript projects that use npm dependencies. We find that 55.88% of the CI build time that is associated with dependency updates is wasted on unused dependencies. The median project allocates 56.09% of its dependency-related CI build time to updates of unused dependencies. Below, we distill lessons for CI stakeholders, bot developers, and researchers.

CI stakeholders (i.e., project maintainers and CI providers) should detect and omit unnecessary builds that are triggered by unused-dependency commits. Observations 1 and 2 show that the projects that are most prone to CI waste spend a substantial amount of their quota of free build minutes on unused-dependency commits. Section 7 also shows that this generates a financial burden for CI consumers who can even exceed the quota of free build minutes when we only consider the CI builds of unused-dependency commits. This waste also contributes to financial costs for the CI provider, as well as increased development and maintenance costs. Our research calls for the attention of CI stakeholders to recognize which dependencies disproportionately generate CI waste and devise strategies to mitigate it. To put our results into action, we propose DEP-SCIMITAR[🐞] to automatically recommend skippable builds to mitigate this waste without requiring a change to dependency specifications. A prototype implementation of DEP-SCIMITAR[🐞] for JavaScript projects is publicly available.¹² Through a simulation of the use of DEP-SCIMITAR[🐞], we demonstrate that it yields substantial benefits, detecting 83% of the wasteful commits, and avoiding 2,342 (68.34%) hours of CI build time that would have otherwise been needlessly expended.

Bot developers should effectively manage CI waste due to unused-dependency commits. Observation 3 shows that bots contribute the vast majority (92.93%) of the CI build time that is wasted on unused dependencies. Our findings shed light on the negative impact that bot-generated updates can have on project maintenance and their over-consumption of CI build resources. We recommend that bot developers incorporate tags, such as [CI skip], into commit messages or pull requests that update unused dependencies. Employing this tagging mechanism can enhance the recognition and screening of such updates, allowing projects to allocate their CI resources to more pressing build requests. Additionally, we recommend that bot developers focus first on cutting down waste when they update development dependencies, since these development dependencies are unlikely to affect production environments [19]. Observation 4 provides further support for this, indicating that a considerable amount of CI waste is linked to specific categories of development dependencies.

Researchers should broaden the scope of the impact of unused dependencies beyond CI build time. Although our study offers perspectives on the impact of unused dependencies on CI, future research should consider other implications that unused dependencies may have on development workflows and project maintenance. For example, our study reveals that unused dependencies are often maintained, e.g., by updating to more recent versions when they become available. This may be because organizations are sensitive to the security risks of using outdated dependencies [50], irrespective of whether the dependency is used. This practice also adds to development overhead, suggesting that exploring this area further could yield beneficial insights.

10 DATA AVAILABILITY

We make our dataset, replication package, and DEP-SCIMITAR[🐞] tool publicly available.^{12,14,13}

REFERENCES

- [1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2020. A machine learning approach to improve the detection of CI skip commits. *Transactions on Software Engineering* (2020).
- [2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2019. Which commits can be CI skipped? *Transactions on Software Engineering* 47 (2019).
- [3] Mahmoud Alfadel, Diego Elias Costa, Mouafak Mkhallalati, Emad Shihab, and Bram Adams. 2020. On the threat of npm vulnerable dependencies in node. js applications. *arXiv preprint arXiv:2009.09019* (2020).
- [4] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28 (2023).
- [5] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. 2023. On the discoverability of npm vulnerabilities in node. js projects. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–27.
- [6] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the use of dependabot security pull requests. In *18th International Conference on Mining Software Repositories (MSR)*.
- [7] Victor R Basili, Lionel C Briand, and Walcélio L Melo. 1996. How reuse influences productivity in object-oriented systems. *Commun. ACM* 39 (1996).
- [8] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20 (2015).
- [9] Arka Bhattacharya. 2014. *Impact of continuous integration on software quality and productivity*. Ph. D. Dissertation. The Ohio State University.
- [10] Christopher Bogart, Christian Kästner, and James Herbsleb. 2015. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *International Conference on Automated Software Engineering Workshop*.
- [11] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–56.
- [12] John Businge, Alexander Serebrenik, and Mark van den Brand. 2012. Survival of Eclipse third-party plug-ins. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 368–377.
- [13] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. Buildfast: History-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *Proceedings of the 35th International Conference on Automated Software Engineering*.
- [14] Ching-Chi Chuang, Luís Cruz, Robbert van Dalen, Vladimir Mikovski, and Arie van Deursen. 2022. Removing dependencies from large software projects: are you really sure?. In *22nd International Working Conference on Source Code Analysis and Manipulation*.
- [15] Filipe Roseiro Cogo, Gustavo A Oliva, and Ahmed E Hassan. 2019. An empirical study of dependency downgrades in the npm ecosystem. *Transactions on Software Engineering* 47 (2019).
- [16] Russ Cox. 2019. Surviving software dependencies. *Commun. ACM* 62, 9 (2019), 36–43.
- [17] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in github for MSR studies. In *18th International Conference on Mining Software Repositories (MSR)*.
- [18] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *International Conference on Software Maintenance and Evolution*. IEEE.
- [19] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*.
- [20] Alexandre Decan, Tom Mens, and Hassan Onsori Delickeh. 2023. On the outdatedness of workflows in the GitHub Actions ecosystem. *Journal of Systems and Software* (2023).
- [21] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24 (2019).
- [22] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. 2020. Detecting and characterizing bots that commit code. In *Proceedings of the 17th international conference on mining software repositories*.
- [23] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *16th International Conference on Mining Software Repositories*.
- [24] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. 2020. Empirical study of restarted and flaky builds on Travis CI. In *Proceedings of the 17th International Conference on Mining Software Repositories*.
- [25] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*.
- [26] Wagner Felidré, Leonardo Furtado, Daniel A Da Costa, Bruno Cartaxo, and Gustavo Pinto. 2019. Continuous integration theater. In *International Symposium on Empirical Software Engineering and Measurement*.

- [27] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing malicious package updates in npm with a lightweight permission system. In *43rd International Conference on Software Engineering (ICSE)*. 1334–1346.
- [28] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. 2020. Accelerating continuous integration by caching environments and inferring dependencies. *Transactions on Software Engineering* (2020).
- [29] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. 2022. Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI. In *Proc. of the International Conference on Software Engineering*.
- [30] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and heterogeneity in historical build data: an empirical study of travis ci. In *Proceedings of the 33rd International Conference on Automated Software Engineering*.
- [31] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. 2019. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering* 24 (2019).
- [32] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, Ying Zou, and Ahmed E Hassan. 2019. Studying the impact of noises in build breakage data. *IEEE Transactions on Software Engineering* 47 (2019).
- [33] Antonios Kkortzis, Daniel Feitosa, and Diomidis Spinellis. 2019. A double-edged sword? Software reuse and potential security vulnerabilities. In *Reuse in the Big Data Era: 18th International Conference on Software and Systems Reuse*.
- [34] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. 2022. On the rise and fall of CI services in GitHub. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
- [35] Foyzul Hassan and Xiaoyin Wang. 2017. Change-aware build prediction model for stall avoidance in continuous integration. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*.
- [36] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *Transactions on Software Engineering* (2023).
- [37] Joseph Hejderup and Georgios Gousios. 2022. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software* 183 (2022).
- [38] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, Darko Marinov, et al. 2016. Continuous integration (CI) needs and wishes for developers of proprietary code. (2016).
- [39] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*.
- [40] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st international conference on automated software engineering*.
- [41] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency smells in Javascript projects. *Transactions on Software Engineering* 48 (2021).
- [42] Lukas Jendele, Markus Schwenk, Diana Cremareno, Ivan Janicijevic, and Mikhail Rybalkin. 2019. Efficient automated decomposition of build targets at large-scale. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*.
- [43] Xianhao Jin and Francisco Servant. 2020. A cost-efficient approach to building in continuous integration. In *Proceedings of the 42nd International Conference on Software Engineering*.
- [44] Xianhao Jin and Francisco Servant. 2022. Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software* 188 (2022).
- [45] Xianhao Jin and Francisco Servant. 2023. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–39.
- [46] Md Mahir Asef Kabir, Ying Wang, Danfeng Yao, and Na Meng. 2022. How Do Developers Follow Security-Relevant Best Practices When Using NPM Packages?. In *Secure Development Conference (SecDev)*.
- [47] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*.
- [48] Igbek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications.. In *RAID*.
- [49] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018).
- [50] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2022. Not All Dependencies are Equal: An Empirical Study on Production Dependencies in NPM. In *37th International Conference on Automated Software Engineering*.
- [51] Rungroj Maipradit, Dong Wang, Patanamon Thongtanunam, Raula Gaikovina Kula, Yasutaka Kamei, and Shane McIntosh. 2023. Repeated Builds During Code Review: An Empirical Study of the OpenStack Community. In *Proc. of*

the International Conference on Automated Software Engineering.

- [52] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*.
- [53] Hamid Mohayjeji, Andrei Agaronian, Eleni Constantinou, Nicola Zannone, and Alexander Serebrenik. 2023. Investigating the resolution of vulnerable dependencies with dependabot security updates. In *20th International Conference on Mining Software Repositories (MSR)*.
- [54] J David Morgenthaler, Misha Gridnev, Raluca Sauciu, and Sanjay Bhansali. 2012. Searching for build debt: Experiences managing technical debt at Google. In *third international workshop on managing technical debt*.
- [55] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* (2017).
- [56] Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. 2023. On the Usage of New JavaScript Features through Transpilers: The Babel Case. *IEEE Software* (2023).
- [57] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement*.
- [58] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*.
- [59] Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. 2021. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *International Conference on Software Maintenance and Evolution*.
- [60] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2020. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology* 128 (2020).
- [61] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
- [62] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The emergence of software diversity in maven central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 333–343.
- [63] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
- [64] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26 (2021).
- [65] Mohsen Vakilian, Raluca Sauciu, J David Morgenthaler, and Vahab Mirrokni. 2015. Automated decomposition of build targets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 123–133.
- [66] Bogdan Vasilescu, Alexander Serebrenik, and Vladimir Filkov. 2015. A Data Set for Social Diversity Studies of GitHub Teams. In *Proceedings of the 12th Working Conference on Mining Software Repositories, Data Track (MSR)*. <https://doi.org/10.1109/MSR.2015.77>
- [67] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 10th joint meeting on foundations of software engineering*.
- [68] Hernan C Vazquez, J Pace, Claudia Marcos, and Santiago Vidal. 2022. Retrieving and Ranking Relevant JavaScript Technologies from Web Repositories. *arXiv preprint arXiv:2205.15086* (2022).
- [69] Pei Wang, Jingiu Yang, Lin Tan, Robert Kroeger, and J David Morgenthaler. 2013. Generating precise dependencies for large software. In *4th International Workshop on Managing Technical Debt (MTD)*.
- [70] Nimmi Weeraddana, Mahmoud Alfadhel, and Shane McIntosh. 2024. Characterizing Timeout Builds in Continuous Integration. *IEEE Transactions on Software Engineering* (2024).
- [71] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2019. A conceptual replication of continuous integration pain points in the context of Travis CI. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
- [72] Timo Wolf, Adrian Schroter, Daniela Damian, and Thanh Nguyen. 2009. Predicting build failures using social network analysis on developer communication. In *31st international conference on software engineering*.
- [73] Ahmed Zerouali, Tom Mens, Jesus Gonzalez-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. 2019. A formal framework for measuring technical lag in component repositories—and its application to npm. *Journal of Software: Evolution and Process* 31 (2019).
- [74] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX security symposium*, Vol. 17.