Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000
DOI: xxx/xxxx

EMPIRICAL

A Formal Framework for Measuring Technical Lag in Component Repositories — and its Application to npm

Ahmed Zerouali*[1,2] | Tom Mens*[2] | Jesus Gonzalez-Barahona[1] | Alexandre Decan[2] | Eleni Constantinou[2] | Gregorio Robles[1]

[1]GSyC/LibreSoft, Universidad Rey Juan Carlos, Madrid, Spain
[2]Software Engineering Lab, University of Mons, Belgium

Correspondence
*Email: ahmed.zerouali@umons.ac.be,
Email: tom.mens@umons.ac.be

Present Address
Software Engineering Lab, Building "Da Vinci", University of Mons, Avenue Maistriau 15, 7000 Mons, Belgium

Summary

Reusable Open Source Software (OSS) components for major programming languages are available in package repositories. Developers rely on package management tools to automate deployments, specifying which package releases satisfy the needs of their applications. However, these specifications may lead to deploying package releases that are outdated or otherwise undesirable because they do not include bug fixes, security fixes, or new functionality. In contrast, automatically updating to a more recent release may introduce incompatibility issues. To capture this delicate balance, we formalise a generic model of *technical lag*, a concept that quantifies to which extent a deployed collection of components is outdated with respect to the *ideal* deployment. We operationalise this model for the npm package manager. We empirically analyze the history of package update practices and technical lag for more than 500K packages with about 4M package releases over a seven-year period. We consider both development and runtime dependencies, and study both direct and transitive dependencies. We also analyze the technical lag of external GitHub applications depending on npm packages. We report our findings, suggesting the need for more awareness of, and integrated tool support for, controlling technical lag in software libraries.

KEYWORDS:
technical lag, software reuse, software repository mining, semantic versioning, empirical analysis

1 | INTRODUCTION

Over the past years, depending on external software components has become a common software development practice, especially in the free, Open Source community[1]. This practice can lead to a significant gain in productivity, due to the ability to reuse complex functionality, rather than implementing it from scratch[2]. To further facilitate this form of reuse, many online platforms have been created for distributions of operating systems (e.g., Linux distributions such as Debian and Ubuntu) and the most prominent programming languages (e.g., npm for JavaScript, MavenCentral for Java, RubyGems for Ruby), totaling millions of reusable components.

While the availability and abundance of those reusable components facilitate building software, it can also cause problems in maintenance and evolution. For example, a recent version of an application may be outdated although not because of its own code, but due to depending on components that were not updated to their latest versions. If this happens, there is a higher risk of having bugs and security issues that may have been already fixed[3]. On the other hand, updating to more recent releases of reusable components is not *for free*, since it might lead to a risk of facing backward incompatible changes[4].

To capture how outdated a deployed software package release is w.r.t. the "ideal" situation, Gonzalez-Barahona *et al.*[5] introduced the concept of "technical lag" as *the increasing lag between upstream development and the deployed system if no corrective actions are taken*. Its goal is to enable developers to decide on a more informed basis whether or not to seize the opportunity of relying on more recent component releases, while taking into account the increased risks that may result from keeping one's dependencies outdated[6]. Technical lag can be measured in many ways. The mechanism of semantic versioning, that allows component maintainers to attach some semantics to their component releases (e.g., whether it is a major release, minor release or patch), helps in letting us define a metric for that lag, based on the version of the releases.

While assessing the problem of technical lag is important at the level of individual components, it becomes more relevant and problematic when large collections of components are involved, such as when an application depending on many components is deployed. For example, if a component imposes version constraints that are too strict on its direct dependencies, it may suffer from higher technical lag. The impact of this lag becomes even more important if transitive dependencies (a component depending on another component, itself depending on other components) are taken into account. However, developers can only act on how they specify direct dependencies for their components, which means that undirect dependencies are, to some extent, beyond their control.

The main objective of this paper is to propose a *formal framework for measuring technical lag*. This framework can be instantiated to specific reusable component repositories, using different ways of measuring lag, in order to study how dependencies and other characteristics of reusable components affect how far away deployments of applications are from the "ideal" deployment. As a validation, we instantiate this formal framework to the case study of the npm repository, a huge library of JavaScript packages that are deployable through the npm package manager, and that form a complex package dependency network. We consider the ideal deployment as having the latest available release for all deployed package releases. We measure the technical lag w.r.t. this ideal deployment in two different ways, based on i) a *time difference* between deployed and ideal releases, and ii) a *version difference* reflecting the number of missed versions.

This paper builds further upon our previous work that analysed technical lag in npm[7]. In it, we offered the first empirical evidence of the presence of technical lag in npm. We observed that a large number of dependencies in npm have a technical lag of several months. In a follow-up work[8], we analyzed the technical lag in npm in more detail. Focusing only on direct runtime dependencies, we explored how technical lag increases over time, taking into account the release type and the use of package dependency constraints. We also explored how technical lag can be reduced by relying on the semantic versioning policy. However, these works did not consider the effect of transitive dependencies on technical lag, and did not study the effect of such lag on external applications relying on npm packages. Moreover, the definitions used for technical lag differed in both papers, and were not based on a formal measurement framework.

In this article, we propose a formal framework for empirically studying how dependencies expressed between npm package releases cause technical lag, how this lag evolves over time, how it propagates over transitive dependencies, and how the lag affects external applications depending on npm packages. In particular, we address the following research questions:

- RQ_0: Which operators are most frequently used in dependency constraints?
- RQ_1: How much technical lag is induced by direct dependencies?
- RQ_2: How do constraint operators impact on technical lag?
- RQ_3: How does technical lag affect external applications?
- RQ_4: How does technical lag propagate over transitive runtime dependencies?

The remainder of this article is structured as follows: Section 2 discusses related work. Section Section 3 introduces a motivating example. Section 4 defines the formal framework for measuring technical lag and instantiates this framework to the case study of the npm package repository. Section 5 empirically studies the research questions for the selected npm case study. Section 6 highlights the novel contributions, discusses our findings, and outlines possible directions for future work, while Section 7 discusses the limitations of this work. Section 8 concludes.

2 | RELATED WORK

Software component library reuse has been an important topic of software engineering research for several decades[9]. Around the 2000s, with the emergence of COTS (components-off-the-shelf), many researchers have investigated how to manage dependencies and evolution[10,11], and have advocated the need to have powerful package managers (often referred to as *configuration management tools*) that allow to improve software reuse[12]. Nowadays, such package managers have become commonplace, due to the rising popularity of Open Source Software repositories, accessible through online collaborative platforms.

The study of software dependencies has been deeply investigated in Linux-based software component distributions[13]. In this regard, researchers have found different types of dependencies that may arise between software components[14], and have proposed solutions for managing dependencies in evolving component distributions[15]. In particular, Abate *et al.*[16] provide a formal framework for analysing the future of software component repositories. They applied this framework to detect future problems related to challenging upgrades and outdated packages, and validated it on the Debian distribution. This approach is quite complementary to the work presented in the current manuscript, which provides a formal framework and associated metrics for studying the temporal evolution of outdated package dependencies.

Several related works have focused on the Maven ecosystem of Java packages. In order to help developers to decide when to use which version of a software library, Mileva et *et al.*[17] proposed an approach and an associated tool based on concept of the "wisdom of the crowds". If a library version is used by more developers, it is more likely to be recommended. The authors acknowledge that other context-specific factors need to be taken into account to recommend the most appropriate version of a library, and that a cost-benefit analysis needs to be made before deciding to switch to a new version of a software library. The authors validated the tool by anlayzing hundreds of Java libraries. Raemaekers *et al.*[18] investigated the usage of semantic versioning by Java packages in *Maven* over a seven-year period. They found that package maintainers did not respect the semantic versioning syntax (e.g., one third of all minor releases introduce at least one breaking change), and that the adherence to semantic versioning only marginally increases over time. Kula *et al.*[3] investigated the latency in adopting the latest library release for thousands of Java libraries. They found that maintainers are less likely to adopt the latest releases at the beginning of a project; and *Maven* libraries are becoming more inclined to adopt the latest releases when introducing new libraries. In a more recent work, Kula *et al.*[19] empirically studied library migration for more than 4,600 GitHub projects and 2,700 library dependencies. They observed that four out of five of the studied projects keep their outdated dependencies. When surveying the project maintainers, they discovered that the large majority of them were unaware of such outdated dependencies. Macho *et al.*[20] proposed the *BuildMedic* tool to automatically repair Maven builds that break due to dependency-related issues.

Dependency-related problems have also been investigated in packaged-based software ecosystems (e.g., npm, CRAN, RubyGems). Decan *et al.*[21] studied the evolution of such huge package dependency networks. Among others, they observed that, in combination with semantic versioning, dependency constraints can prevent packages from breaking due to dependency updates. Abdalkareem *et al.*[22] focused on potential problems caused by the huge number of dependencies on "trivial" packages in npm. While interviewed developers did not consider those packages as harmful, they were found to be less tested than other packages. Decan *et al.*[1] compared the evolution of the npm package dependency network with six other packaging ecosystems, and compared their growth, changeability, reusability and fragility over time. The high and increasing number of transitive dependencies was found to be a major cause of fragility, suggesting the need for better dependency management tools and policies.

Several researchers found that outdated dependencies are a potential source of security vulnerabilities. Cox *et al.*[6] analyzed 75 Java projects that manage their dependencies through *Maven* and introduced different metrics to quantify their use of recent versions of dependencies. They observed that systems using outdated dependencies were four times more likely to have security issues and backward incompatibilities than systems that are up-to-date. Lauinger et al.[23] studied the client-side use of JavaScript libraries. They found that "the time lag behind the newest release of a library is measured in the order of years" and that this is a major source of known vulnerabilities in websites using these libraries. They also observed that "libraries included transitively [...] are more likely to be vulnerable". Decan *et al.*[24] carried out an empirical analysis of security vulnerabilities in the npm repository by analyzing how and when these vulnerabilities are discovered and fixed, and to which extent they affect other package releases in the repository in presence of dependency constraints. They observed that it often takes a long time to discover vulnerabilities since their introduction. A non-negligible proportion of vulnerabilities (15%) are considered to be of high risk since they are either fixed after public announcement of the vulnerability, or not fixed at all. They found the presence of package dependency constraints to play an important role in fixing vulnerabilities, mainly because the imposed dependency constraints prevent more recent releases to be installed.

All of the above works illustrate that component reuse can suffer from outdated dependencies that may have important consequences such as backward incompatibilities and security vulnerabilities, often without even being aware of them. This highlights the need for measuring the *technical lag* of package releases and their dependencies, which is the scope of the current paper. González-Barahona *et al.*[5] introduced the theoretical concept of *technical lag* for the first time, to measure how outdated deployed software systems are. They suggested many ways in which technical lag can be implemented and could be used when deciding about upgrading in production. They also illustrated the evolution of technical lag for some specific packages in the Debian Linux distribution. Zerouali *et al.*[7] analyzed technical lag for package dependencies in the npm repository, in order to assess how outdated a software package is with respect to the latest available releases of its direct dependencies. They observed a strong presence of technical lag, and attributed this lag to the specific use of dependency constraints. Decan *et al.*[8] carried out a more detailed analysis of the evolution of such technical lag induced by direct dependencies in the npm repository. They observed that many npm package releases exhibit technical lag, and assessed to which extent a better use of semantic versioning and dependency constraints would allow to reduce such lag, allowing dependent packages to benefit from fixes to vulnerabilities in minor or patch releases in dependent packages.

This paper generalises and extends upon these works, by introducing a formal framework of technical lag. We conduct an empirical study using an instantiation of this framework to the npm package dependency network. We assess the evolution of technical lag of (collections of) npm package releases and their associated package dependencies in terms of a time difference and version difference. We compare the difference observed for runtime dependencies and deployment dependencies. We also assess the technical lag induced by *transitive* dependencies. Finally, we analyze the impact of technical lag beyond the boundaries of the npm repository, by considering external applications depending on npm packages.

3 | MOTIVATING EXAMPLE

To illustrate the need for a formal framework for measuring technical lag, we start by presenting a concrete, motivating example, based on the npm package repository. In order to better understand the example, we first present the main ideas of semantic versioning and version constraints.

3.1 | Semantic versioning and version constraints

Semantic Versioning (henceforth referred to as *semver*[1]) has become a popular policy to recommend how to assign and increment version numbers of new component releases. It provides a simple set of rules and requirements to communicate the type of changes made when releasing a new version of a software component. This allows dependent software components to be informed about possible "breaking changes"[25]. A *semver*-compatible release uses a version number composed of a *major*, *minor* and *patch* number. This format allows to order releases and indicates the importance of each new release. For example, *1.2.3* occurs before *1.2.10* (higher patch number), which occurs before *1.3.0* (higher minor number), which occurs before *2.1.0* (higher major number). Backward incompatible updates should increment the *major* number, backward compatible updates that respect the API but may add new functionalities should increment the *minor* number, while simple bug fixes or security patches should increment the *patch* number.

A component can restrict the releases of other components on which it depends by specifying version constraints in the specification of the dependencies. Upon installation of the component, the installation manager will consider these constraints to install the most "appropriate release" for each dependency. For example, the npm package manager will select for installation the latest available release satisfying the dependency constraints. Version constraints can be quite diverse. Table 1 summarizes the types of version constraints that can be used for specifying npm package dependencies, together with the interpretation of each constraint type.

Constraint	Interpretation	Notation (example)	Satisfied versions
strict	use exactly this version	2.0.0	2.0.0
tilde (~)	use the latest available *patch* update	$\sim$ 2.3.0	$\geq$ 2.3.0 $\wedge$ $<$2.4.0
caret (^)	use the latest available *minor* or *patch* update	^2.3.0	$\geq$2.3.0 $\wedge$ $<$3.0.0
latest	use the latest available release	latest, x, x.x.x, $*$ or $*.*.*$	$\geq$0.0.0
minimal	use the latest available release above this version	$>=$ 2.3.0	$\geq$ 2.3.0
maximal	use the latest available release below this version	$<$ 2.3.0	$<$ 2.3.0
version ranges	use the latest available release in the specified version interval	1.2.3 - 2.3.4	$\geq$1.2.3 $\wedge$ $\leq$2.3.4
logic operators	any logic combination of constraints	2.5.3 || $>=$2.8.1	2.5.3 $\vee$ 2.8.1 $\vee$ $>$2.8.1
wildcard	allow updates to any release compatible with the wildcard	1.2.x	$\geq$1.2.0 $\wedge$ $<$1.3.0
		2.*	$\geq$2.0.0 $\wedge$ $<$3.0.0

TABLE 1 Types of dependency constraints for npm package dependencies.

3.2 | Example of a package dependency tree

The main purpose of the example is to illustrate how and why component releases can become outdated over time, and to illustrate different ways to quantify the extent to which component releases can be outdated.

Consider the actual software package **youtube-player** taken from the npm repository. Figure 1 shows the dependency tree of package release **youtube-player 5.5.0** at its release date. Its direct and transitive dependencies are shown in black, and correspond to all other packages that will have a release installed when the user decides to run **npm install** for **youtube-player 5.5.0** at its release date. The exact release that will be installed for each dependency is determined by the dependency constraint, shown in Figure 1 on the edges of the tree. For example, the constraint ^**2.6.6** on **debug** specifies that the latest minor or patch release above version 2.6.6 will be selected upon installation. At the release date of **youtube-player 5.5.0**, constraint ^**2.6.6** on **debug** will select **debug 2.6.9** for installation. Figure 1 also shows some more recent releases in red. Even if they are more recent, these releases will not be selected for installation because they do not satisfy the dependency constraints. For example, **debug**

[1]https://semver.org and https://docs.npmjs.com/misc/semver

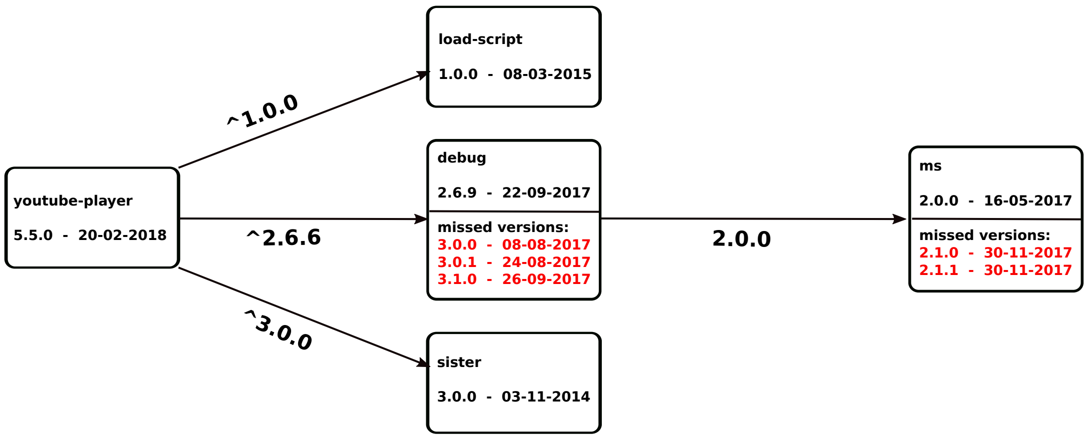


FIGURE 1 Transitive dependencies of version **5.5.0** of the **youtube-player** package at its release date of 2018-02-02.

3.0.0, **3.0.1** and **3.1.0** are not accepted by the constraint ^**2.6.6**. It is because of such non-installable, more recent releases that **youtube-player 5.5.0** could be considered as outdated with respect to a more "ideal" situation were the highest release of each dependency is installed.

3.3 | Technical lag as a time difference

The notion of technical lag aims to capture the difference (or "delta") between the current situation and an "ideal" one. Assuming that we measure the technical lag as a time difference, the lag induced by not using the latest version of **debug** would be **4 days** (namely the delta between the release date of **debug 2.6.9** and **debug 3.1.0**). If we assume that the aggregated time lag of **youtube-player 5.5.0** is computed as the maximum of the lags induced by all its direct dependencies, the lag will remain 4 days, because the two other dependencies on packages (**sister** and **load-script**) are up-to-date. If we also take transitive dependencies into account, we need to consider the lag induced by package releases at deeper levels of the dependency tree as well. Package release **ms 2.0.0** induces a technical lag because the latest available release **ms 2.1.1** is not accepted by the strict dependency constraint **2.0.0**. Because of this, **ms 2.0.0** has a technical lag of **167 days**, corresponding to the time delta between the release date of **ms 2.0.0** and **ms 2.1.1**. Consequently, the aggregated (maximum) time lag of the entire (transitive) dependency tree for **youtube-player 5.5.0** is **167 days**.

3.4 | Technical lag as a version difference

Although the semver policy is not always respected by package maintainers[18], it provides a way of telling apart releases, and provides relevant information on which upgrades of dependent releases are more likely to cause breaking changes. We will therefore rely on semantic version numbers to measure the version lag between npm package releases.

Reconsidering our motivating example, the sequence of available package releases (at the release date of **youtube-player 5.5.0**) since **debug 2.6.9** is $[2.6.9, 3.0.0, 3.0.1, 3.1.0]$. This sequence can be used to count the number of missed updates between the selected one (**2.6.9**) and the highest available one (**3.1.0**), considering each version component separately. For this particular example, the version delta will be 1 major (the change from version 2.6.9 to 3.0.0) + 1 minor (the change from version 3.0.1 to 3.1.0) + 1 patch (the change from version 3.0.0 to 3.0.1). In a similar way, the version delta between the selected package release of **ms** (**2.0.0**) and the highest available one (**2.1.1**), based on the sequence $[2.0.0, 2.1.0, 2.1.1]$, will be 0 major + 1 minor (the change from 2.0.0 to 2.1.0) + 1 patch (the change from 2.1.0 to 2.1.1). Based on the above, if we would compute the aggregated version lag of **doc-prsr 2.1.1** as the sum of the version lags induced by all its transitive dependencies, we would obtain a lag of : (1 major, 1 minor, 1 patch) + (0 major, 1 minor, 1 patch) = **(1 major, 2 minor, 2 patch)**.

4 | A FORMAL FRAMEWORK FOR TECHNICAL LAG

The motivating example of Section 3 has illustrated how to compute and aggregate technical lag, reflecting the extent to which a component release is outdated, in many different ways. It is the purpose of the technical lag framework, proposed in this section, to capture and formalise all these variations and future variants. In order to measure technical lag in any given repository of reusable software components, we need a formal framework that abstracts the specificities of the various repositories. Refining the definition of Gonzalez-Barahona *et al.*[5], we define technical lag for a certain component release as the *difference* between that release and the *ideal* release, where *ideal* could be interpreted in different ways: most recent, most compatible, most stable, most secure, etc. Then, we can apply the concept to a collection of component releases, as the aggregated lag for all of them with respect to the ideal releases for those components. A specific situation where it is useful to apply this definition for a collection of releases is when we install a certain component release together with the collection of all its direct or transitive dependencies.

4.1 | Technical lag framework

To formally capture the notion of technical lag in a generic way, we define a parameterised technical lag framework:

Definition 1. Technical lag framework
A *technical lag framework* is a tuple $\mathcal{F} = (\mathcal{C}, \mathcal{L}, \mathbf{ideal}, \mathbf{delta}, \mathbf{agg})$ where

- $\mathcal{C}$ is a set of component releases.
- $\mathcal{L}$ is a set of possible lag values.
- $\mathbf{ideal} : \mathcal{C} \to \mathcal{C}$ is a function returning the "most preferred" component release (according to the user desiring to deploy a component) over a given one.
- $\mathbf{delta} : \mathcal{C} \times \mathcal{C} \to \mathcal{L}$ is a function computing the difference (in terms of lag induced) between a first component release and a second one.
- $\mathbf{agg} : \mathbb{P}(\mathcal{L}) \to \mathcal{L}$ is a function aggregating the results of a set of lag values[2]. Typical examples of $\mathbf{agg}$ functions would be the sum, maximum, mean or median of a set of values.

Given a technical lag framework $\mathcal{F}$, we can formally define the *technical lag* induced by choosing a component release instead of the $\mathbf{ideal}$ (i.e., most preferred) release for that component. What this means may differ depending on the considered scenario (e.g., we may wish to create the most stable, most recent, or most secure deployment). If we use a component release instead of the ideal one, the induced technical lag is the difference ($\mathbf{delta}$) between both releases.

Definition 2. Technical lag
$\mathbf{techlag}_{\mathcal{F}} : \mathcal{C} \to \mathcal{L}$ such that $\mathbf{techlag}_{\mathcal{F}}(\mathsf{c}) = \mathbf{delta}(\mathsf{c}, \mathbf{ideal}(\mathsf{c}))$

Similarly, we can define the *aggregated technical lag* induced by a set of component releases $\mathsf{D} \subseteq \mathcal{C}$.

Definition 3. Aggregated technical lag
$\mathbf{agglag}_{\mathcal{F}} : \mathbb{P}(\mathcal{C}) \to \mathcal{L}$ such that $\mathbf{agglag}_{\mathcal{F}}(\mathsf{D}) = \mathbf{agg}(\{\mathbf{techlag}_{\mathcal{F}}(\mathsf{c}) \mid \forall \mathsf{c} \in \mathsf{D}\})$

4.2 | Instantiating the framework to the npm case study

The technical lag framework has been designed to be generally applicable to different types of component distributions, and even in different ways for a given component distribution. We illustrate this by instantiating the framework to the case study of the npm repository of JavaScript package releases.

Definition 4. Package releases
Let $\mathcal{P} \subset \mathcal{N} \times \mathcal{V} \times \mathcal{T}$ be the set of all package releases available in npm, where $\mathcal{N}$ is the set of all possible package names, $\mathcal{V} \subset \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ the set of all possible version numbers, and $\mathcal{T}$ the set of all possible time points. Each package release $\mathsf{p} = (\mathsf{p}_{\mathsf{name}}, \mathsf{p}_{\mathsf{version}}, \mathsf{p}_{\mathsf{time}}) \in \mathcal{P}$ has an associated package name $\mathsf{p}_{\mathsf{name}} \in \mathcal{N}$, a version number $\mathsf{p}_{\mathsf{version}} \in \mathcal{V}$ and a release date $\mathsf{p}_{\mathsf{time}} \in \mathcal{T}$. We assume a total order on $\mathcal{V}$ and $\mathcal{T}$.

We propose multiple instantiations (i.e., scenarios of use) of the framework that mainly differ in how technical lag is computed. In a first scenario, technical lag is computed as a time difference between two releases. In a second scenario, technical lag is computed based on a difference between version numbers of two releases. Other scenarios could be envisaged to explore a wider spectrum of ways to compute technical lag, but this is outside of the scope of the current paper.

[2]The notation $\mathbb{P}(\mathcal{L})$, which can alternative be written as $2^{\mathcal{L}}$, represents the powerset of $\mathcal{L}$, i.e., the set of all possible subsets of $\mathcal{L}$.

Definition 5. Time-based instantiation of the technical lag framework

For any given time point $\mathsf{t} \in \mathcal{T}$ we define $\mathcal{F}^{\mathsf{npm}}_{\mathsf{time}}(\mathsf{t})$ as the function that instantiates the technical lag framework $\mathcal{F}$ at time t, as follows:

$$\mathcal{F}^{npm}_{time}(t) = \left(\mathcal{P}_t, \mathcal{T}, \mathbf{ideal}^{npm}, \mathbf{delta}^{npm}_{time}, \mathbf{agg}_{time}\right)$$

where:

- $\mathcal{P}_\mathsf{t} = \{\mathsf{p} \in \mathcal{P} \mid \mathsf{p}_{\mathsf{time}} \leq \mathsf{t}\}$ is the set of npm package releases that are available at time t.
- $\mathbf{ideal}^{\mathsf{npm}}(\mathsf{p}) = \max_{\mathsf{p}'_{\mathsf{version}}} \{\mathsf{p}' \in \mathcal{P}_\mathsf{t} \mid \mathsf{p}'_{\mathsf{name}} = \mathsf{p}_{\mathsf{name}}\}$. Intuitively, the function $\mathbf{ideal}^{\mathsf{npm}}$ mimicks the choice of the npm package manager, i.e., for a given package release p it returns the *highest available version*[3] of a package release with the same name as p.

  (An alternative variant of $\mathbf{ideal}^{\mathsf{npm}}$ is to select the *highest backward compatible version* for a given package release, restricting the selection of higher releases to those corresponding to the same major release number only.)
- $\mathbf{delta}^{\mathsf{npm}}_{\mathsf{time}}(\mathsf{p}, \mathsf{q}) = \max(0, \mathsf{q}_{\mathsf{time}} - \mathsf{p}_{\mathsf{time}})$ computes the positive difference between the release dates of two package releases p and q.
- $\mathbf{agg}_{\mathsf{time}}(\mathsf{L}) = \max(\mathsf{L})$, with $\mathsf{L} \subseteq \mathcal{T}$ computes the maximum of a collection of time points.

We can directly compute $\mathbf{techlag}_{\mathcal{F}^{\mathsf{npm}}_{\mathsf{time}}(\mathsf{t})}(\mathsf{p})$ in terms of the above framework instantiation. This definition formalises the time lag that we informally explained with the motivating example in Section 3.

Note that the aggregation function chosen for $\mathbf{agg}_{\mathsf{time}}$ (maximum) is useful to ascertain that the lag of each release in the collection remains below a certain threshold. Alternatively, we could use a summation to assess the total lag for the collection as a whole, as an estimation of the effort that would be needed to reduce the lag in all releases of the collection.

We now define a version-based instantiation of the technical lag framework for npm, corresponding to the second scenario:

Definition 6. Version-based instantiation of the technical lag framework

For any given time point $\mathsf{t} \in \mathcal{T}$ we define $\mathcal{F}^{\mathsf{npm}}_{\mathsf{version}}(\mathsf{t})$ as the function that instantiates the technical lag framework $\mathcal{F}$ at time t, as follows:

$$\mathcal{F}^{npm}_{version}(t) = \left(\mathcal{P}_t, \mathcal{V}, \mathbf{ideal}^{npm}, \mathbf{delta}^{npm}_{version}, \mathbf{agg}_{version}\right)$$

where:

- $\mathbf{ideal}^{\mathsf{npm}}$ is defined in the same way as for $\mathcal{F}^{\mathsf{npm}}_{\mathsf{time}}(\mathsf{t})$
- $\mathbf{delta}^{\mathsf{npm}}_{\mathsf{version}}(\mathsf{p}, \mathsf{q})$ is defined as $(\mathsf{major}_{\mathsf{lag}}, \mathsf{minor}_{\mathsf{lag}}, \mathsf{patch}_{\mathsf{lag}})$ where
  $\mathsf{coll} = \{\mathsf{r} \in \mathcal{P}_\mathsf{t} \mid \mathsf{p}_{\mathsf{time}} \leq \mathsf{r}_{\mathsf{time}} \leq \mathsf{q}_{\mathsf{time}} \wedge \mathsf{p}_{\mathsf{name}} = \mathsf{r}_{\mathsf{name}} = \mathsf{q}_{\mathsf{name}}\}$
  $\mathsf{major}_{\mathsf{lag}} = |\{\mathsf{major} \mid \mathsf{r}_{\mathsf{version}} = (\mathsf{major}, \mathsf{minor}, \mathsf{patch}), \forall \mathsf{r} \in \mathsf{coll}\}| - 1$
  $\mathsf{minor}_{\mathsf{lag}} = |\{(\mathsf{major}, \mathsf{minor}) \mid \mathsf{r}_{\mathsf{version}} = (\mathsf{major}, \mathsf{minor}, \mathsf{patch}), \forall \mathsf{r} \in \mathsf{coll}\}| - \mathsf{major}_{\mathsf{lag}} - 1$
  $\mathsf{patch}_{\mathsf{lag}} = |\{(\mathsf{major}, \mathsf{minor}, \mathsf{patch}) \mid \mathsf{r}_{\mathsf{version}} = (\mathsf{major}, \mathsf{minor}, \mathsf{patch}), \forall \mathsf{r} \in \mathsf{coll}\}| - \mathsf{major}_{\mathsf{lag}} - \mathsf{minor}_{\mathsf{lag}} - 1$
  In this definition, coll is the set of all releases between p and q. We then find the number of releases that increase the major number, then the number of releases that increase the minor number while having the same major number, and finally the number of patch releases that increase the patch number while having the same major and minor numbers. From that, we derive the number of distinct major, minor and patch releases that would be needed to upgrade from r_1 to r_2.
- $\mathbf{agg}_{\mathsf{version}}(\mathsf{L}) = \sum_{\mathsf{v} \in \mathsf{L}} \mathsf{v}$, with $\mathsf{L} \subseteq \mathcal{V}$ computes the sum over a set of versions. To do so, addition has to be defined on versions numbers, e.g., as follows. Let $\mathsf{v}, \mathsf{w} \in \mathcal{V}$ be two version numbers. Assume $\mathsf{v} = (\mathsf{v}_{\mathsf{major}}, \mathsf{v}_{\mathsf{minor}}, \mathsf{v}_{\mathsf{patch}})$ and $\mathsf{w} = (\mathsf{w}_{\mathsf{major}}, \mathsf{w}_{\mathsf{minor}}, \mathsf{w}_{\mathsf{patch}})$, then $\mathsf{v} + \mathsf{w} = (\mathsf{v}_{\mathsf{major}} + \mathsf{w}_{\mathsf{major}}, \mathsf{v}_{\mathsf{minor}} + \mathsf{w}_{\mathsf{minor}}, \mathsf{v}_{\mathsf{patch}} + \mathsf{w}_{\mathsf{patch}})$.

We can directly compute $\mathbf{techlag}_{\mathcal{F}^{\mathsf{npm}}_{\mathsf{version}}(\mathsf{t})}(\mathsf{p})$ in terms of the above definitions. This formalises the version lag that we informally explained with the motivating example in Section 3.

Notation 1. In the remainder of this paper we use the shortcut notations $\mathbf{techlag}_{\mathsf{time}}(\mathsf{p}, \mathsf{t})$ for $\mathbf{techlag}_{\mathcal{F}^{\mathsf{npm}}_{\mathsf{time}}(\mathsf{t})}(\mathsf{p})$ and $\mathbf{agglag}_{\mathsf{time}}(\mathsf{p}, \mathsf{t})$ for $\mathbf{agglag}_{\mathcal{F}^{\mathsf{npm}}_{\mathsf{time}}(\mathsf{t})}(\mathsf{p})$, and similarly for the version-based variants $\mathbf{techlag}_{\mathsf{version}}$ and $\mathbf{agglag}_{\mathsf{version}}$, as it is clear from the context that we are referring to the npm instantiation of the formal framework.

[3]assuming that the user does not specify an additional upper bound version constraint upon installation of p

In practice, when installing an npm package release, *dependency relationships* impose the installation of all required package releases as well. These package releases are selected by the `npm install` tool upon installation of a given package.

Definition 7. Direct and transitive dependencies

Let $\mathtt{t} \in \mathcal{T}$ be a point in time. Let $\mathcal{P}_\mathtt{t}$ be the set of npm package releases available at time $\mathtt{t}$. We define the two following functions:

$\mathbf{deps}_\mathtt{t} : \mathcal{P}_\mathtt{t} \to \mathbb{P}(\mathcal{P}_\mathtt{t})$ such that $\mathbf{deps}_\mathtt{t}(\mathtt{p})$ returns all npm package releases satisfying the direct dependencies of $\mathtt{p}$, as selected by `npm install`.

$\mathbf{deps^+}_\mathtt{t} : \mathcal{P}_\mathtt{t} \to \mathbb{P}(\mathcal{P}_\mathtt{t})$ such that $\mathbf{deps^+}_\mathtt{t}(\mathtt{p})$ returns all npm package releases satisfying the *transitive* (i.e., direct and indirect) dependencies of $\mathtt{p}$, as selected by `npm install`. Alternatively, we can define $\mathbf{deps^+}_\mathtt{t}(\mathtt{p})$ as the minimal fix point such that:

$$\mathbf{deps}_t(p) \subseteq \mathbf{deps^+}_t(p) \quad \text{and} \quad \forall p' \in \mathbf{deps^+}_t(p) : \mathbf{deps}_t(p') \subseteq \mathbf{deps^+}_t(p)$$

Based on these two functions and on the definitions of $\mathbf{agglag}_\alpha$ for $\alpha \in \{\text{time}, \text{version}\}$ in the technical lag framework, we can define the technical lag of a deployment of a package.

Definition 8. Technical lag of a package deployment

Let $\mathtt{t} \in \mathcal{T}$ be a point in time. Let $\mathcal{P}_\mathtt{t}$ be the set of npm package releases available at time $\mathtt{t}$. For $\alpha \in \{\text{time}, \text{version}\}$, we define:

- $\mathbf{deplag}_\alpha(\mathtt{p}, \mathtt{t}) = \mathbf{agglag}_\alpha(\mathbf{deps}_\mathtt{t}(\mathtt{p}), \mathtt{t})$ for the direct dependencies
- $\mathbf{deplag}^+_\alpha(\mathtt{p}, \mathtt{t}) = \mathbf{agglag}_\alpha(\mathbf{deps^+}_\mathtt{t}(\mathtt{p}), \mathtt{t})$ for the transitive dependencies

For example, $\mathbf{deplag_{time}}(\mathtt{p}, \mathtt{t})$ computes the *time lag* of all *direct dependencies* of package release $\mathtt{p}$ at time $\mathtt{t}$ as the maximum time lag of any of these direct dependencies. Similarly, $\mathbf{deplag^+_{version}}(\mathtt{p}, \mathtt{t})$ computes the *version lag* of all *transitive dependencies* of package release $\mathtt{p}$ at time $\mathtt{t}$ as the sum of all version lags of all these transitive dependencies.

5 | EMPIRICAL ANALYSIS OF TECHNICAL LAG IN npm

Using the instantiation of the technical lag framework to npm, as presented in Section 4.2, we have carried out an empirical analysis of the presence and evolution of technical lag in the npm package repository. All code and data required to reproduce the empirical analysis in this section are available on https://doi.org/10.5281/zenodo.1420075.

5.1 | Characteristics of the npm case study

npm is the most used package manager for reusing JavaScript components, along with its “official” package repository with a large and active developer community[26]. JavaScript is one of the most popular programming languages nowadays. According to the Octoverse study[4]), JavaScript was reported as the most popular programming language on GitHub in 2017. According to Tiobe's programming language index[5] (September 2018), JavaScript is the 8th most popular programming language. npm and JavaScript have also been used as case studies by many other researchers in empirical software engineering[1,22,23,27,28].

To study npm we relied on the dataset from libraries.io[6], that monitors several parameters for 3.3M packages from 36 different package repositories, including npm[7]. For our empirical study we used version 1.2.0 of the Libraries.io Open Source Repository and Dependency Metadata[29], available as open access under the *CC Share-Alike 4.0* license. The considered timeframe for our analysis was from *2010-11-09* (the date of the first known npm package release) to *2018-03-13* (the date of publication of the dataset). The dataset comprises 698K npm packages, 4.76M package releases and 52.8M npm dependencies.

Relevant information for each package release includes the package name, version number, release date, and information for each dependency of the package release, such as the name of the required package, a dependency constraint and a dependency type. The metadata of npm package releases is stored in a `package.json` file[8] that is available for each release. Figure 2 provides an excerpt of relevant information stored in such a file.

npm considers different dependency types, the ones that npm packages make use of are: i) *Runtime* dependencies are required to install and execute the package; ii) *Development* dependencies are used during package development (e.g., for testing); and iii) *Optional* dependencies will not hamper the package from being installed if the dependency is not found or cannot be installed. The rest are *Peer* and *Bundled* dependencies.

[4]octoverse.github.com
[5]https://www.tiobe.com/tiobe-index/
[6]https://libraries.io/about
[7]These numbers correspond to the state of libraries.io as of September 1st, 2018.
[8]The structure of this file is defined in https://docs.npmjs.com/files/package.json, visited on Sept. 18th 2018.

```
{"name": "foo",
 "version": "1.2.3",
 ...
 "dependencies": {"bar" : ">=1.0.2 <2.1.2", "baz" : ">1.0.2 <=2.3.4"},
 "devDependencies": {"boo" : "2.0.1"},
 ...}
```

FIGURE 2 Excerpt of relevant metadata stored in a hypothetical *package.json* file for package release foo 1.2.3.

The dataset is composed of 42.6% of runtime dependencies, 57.3% of development dependencies, and less than 1% (355 in total) of optional dependencies. Because of this very low number of *Optional* dependencies, and because we do not know which of them are actually installed in practice (since they are optional), we have excluded them from our analysis.

Through a careful manual inspection of the dataset, we found a number of packages that should be ignored for the analysis, because they would introduce bias in the results. We are primarily interested in packages that can be considered as reusable libraries, i.e., packages that have been stored on npm for the purpose of being reused by other npm packages or external applications. For this reason, we excluded a huge set of packages that were created automatically[9] on 30th of March 2016 with the only purpose of depending on a large set of npm runtime dependencies. Examples are the "wowdude-x" packages[10], "neat-x" packages[11] and "all-packages-x" packages. We found that a large number of packages (corresponding to more than 20K package releases) were released in May 2017 with a name ending with "-cdn", (e.g., react-native-cdn[12], webpack-cdn[13], etc). All of these packages were removed from later versions of npm, because they were considered as "spam".

We also removed all package pre-releases from the dataset (e.g., releases with version numbers of the form *1.0.0-alpha*, *1.0.0-alpha.1*, *2.1.0-beta*, *2.0.0-rc1* and so on). The reason for this exclusion is that prereleases are not recommended to be reused by a dependency because such releases are unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version [14].

After filtering out the above packages and releases, we obtained a sanitized dataset containing 520K packages, 3.6M package releases, and 46M dependencies for these releases. 57.9% of these dependencies were *runtime dependencies*, while 42.1% were *development dependencies*. We could parse and identify 98.1% of the packages expressed by these dependencies. The other 1.9% are dependencies with local files, git URLs, unknown (e.g., misspelled), etc.

5.2 | Research questions

Based on the formal framework for technical lag measurement and its instantiation for the npm case study (presented in Section 4), we will empirically study five research questions for the npm package repository. We take into account the semver mechanism and the use of version constraints on package dependencies specified by npm package releases.

RQ_0: Which operators are most frequently used in dependency constraints?

Although the npm community encourages developers to use the semver standard in order to have more reliable and predictable updates [15], it is not clear to which extent they are following this recommendation. Therefore, we studied how npm package maintainers are using constraints for their dependencies.

We quantified the usage of the different types of version constraints used by package dependencies defined in npm package releases. In the case of constraints with a combination of other constraints, we consider the type of the less permissive one. This occured in only 0.0005% of all dependencies. Table 2 shows the proportion of the types of dependency constraints used during the considered observation period. We observe that caret is most frequently used, covering over 71.2% (32M) of all dependency constraints. This suggests that most package maintainers want to avoid backward incompatible changes, but aspire to benefit from future bug fixes (patch updates) and new functionalities (minor updates).

[9] https://github.com/ell/npm-gen-all
[10] https://libraries.io/search?q=wowdude
[11] https://libraries.io/npm/neat-106
[12] https://libraries.io/npm/react-native-cdn
[13] https://libraries.io/npm/webpack-cdn
[14] https://semver.org/#spec-item-9
[15] https://docs.npmjs.com/getting-started/semantic-versioning

Dependencies (total)	caret	strict	tilde	latest	other
Runtime (42,3%)	67.5%	16%	8.2%	4.0%	4.3%
Development (57,7%)	74.0%	13.4%	6.9%	3.5%	2.2%
All dataset (100%)	71.2%	14.5%	7.5%	3.7%	3.1%

TABLE 2 Proportion of the types of dependency version constraints used, for all npm package releases over the considered period.

14.5% (6.5M) of all dependencies were specified with a strict version number, signifying that package maintainers prefer a possibly older but fixed version of a dependency, rather than benefiting from automatic updates. Anecdotal evidence suggests that this is often the case: *"I personally don't care about bug fixes. I like of course using a bug-free software but I prefer to rely on stable libraries and if I hit a bug I'll check if it is fixed, in which versions and what this version comes up with. Many times I got a new release downloaded fixing a bug in a feature that I never used. So, I prefer strict versioning where I install the same version every time. Version that I know it works."*[16]

7.5% (3.3M) of all dependencies used a tilde constraint to allow patch updates, and 3.7% (1.7M) of the dependencies are very permissive, allowing to use the latest available version of their dependency. All other types of dependency constraints combined (e.g., comparison operators, wildcards like 1. *. *, etc) represent only 3.1% (1.4M) of all dependencies.

The findings above are mixed. The majority of developers are interested in keeping some backward compatibility, benefiting from minor and patch releases, but do not want to worry about new major releases with incompatible changes, requiring a higher maintenance effort. However, many developers still seem to prefer to keep all control, by imposing a strict version constraint (14.5% of the dependencies). This will likely cause a higher technical lag for many releases in npm, as will be shown later.

From Table 2 we also observe that, while caret constraint is more widely used for development dependencies, tilde, strict and latest constraints are more widely used for runtime dependencies. This could simply be a consequence of the fact that development dependencies are not needed to run packages but only to develop them, developers might care less about managing them and more often use the "default" caret constraint provided by npm.

In previous work[7], we analysed the dynamics of npm dependencies, observing that new releases tend not to remove or add dependencies, but to adapt the version constraints of existing ones. We found that adding or removing dependencies is mostly done during major releases. In order to gain more insight about the kind of releases that change dependency constraints, we quantified when and how version constraints are changed to another type (e.g., from ∼1.0.0 to ^1.0.0) or updated to another constraint of the same type (e.g., from ^1.0.0 to ^2.0.0).

Figure 3 shows box plots of the distribution of the number of dependency constraints that are changed in new major, minor or patch releases. We observe that the third quartile of the distribution is highest for major, and lowest for patch releases. We observe a median value of 1, 2 and 2 changed version constraints for patch, minor and major releases, respectively. This suggests that there is higher chance to update dependencies during minor and major releases, when a feature is added or an incompatible change happened. This corresponds to what is advised by semver [17].

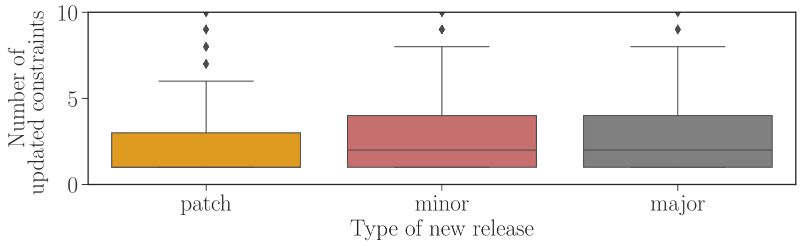


FIGURE 3 Distribution of the number of dependency constraints that are changed in new major, minor or patch releases.

To study the technical lag, we should also know how often packages release new versions. A package with frequent releases may be a real issue for other packages that rely on it, since it could result in a higher than average technical lag for the dependent packages (since they are required to update frequently if they want to keep pace with this release rate). On the other hand, releasing new package versions is important and breaking changes are generally expected for progress because they will enable new features, new requirements, less technical debt, improved performance,

[16]http://krasimirtsonev.com/blog/article/thoughts-on-semantic-versioning-npm-and-JavaScript-ecosystem
[17]https://semver.org/#what-should-i-do-if-i-update-my-own-dependencies-without-changing-the-public-api

and fixed bugs[30]. Such backward incompatible changes are generally considered acceptable if they are clearly signaled as such, for example by increasing the major release number of the semver policy[31,25].

Considering all types of releases of all packages, we found that 14.7% (0.7M) are *initial* releases, i.e., packages released in npm for the first time. 68.8% (3.3M) are *patch* releases, 13% (0.62M) are *minor* releases and only 3.5% (0.17M) are *major* releases. The total is 4.8M releases at the dataset creation date.

Releasing a new package version requires time and effort. Therefore, we studied the time needed to release a new chronological, but not necessarily successive, version of a given type. For example, considering the next series of versions (1.0.0 (initial), 1.0.1 (patch), 1.0.2 (patch), 1.1.0 (minor), 1.1.1 (patch)), and starting with the (initial $\rightarrow$ patch) case, we calculate the time between (1.0.0, 1.0.1). For the case of (initial $\rightarrow$ minor), we calculate the time between (1.0.0, 1.1.0). In a similar way we proceed for the (patch $\rightarrow$ patch) case, we calculate the time between (1.0.1, 1.0.2) and (1.0.2, 1.1.1). For the case of (patch $\rightarrow$ minor) we calculate the time between (1.0.2, 1.1.0), etc.

Figure 4 shows the distribution of days until the next chronological version is released. As can be seen, it takes more time to release a new major version than a minor or patch one. More specifically, on average, it takes about 19 days to release a patch version, 57 days to release a minor version and 120 days to release a major one. The fact that major releases are rare complies with the results of a recent survey among 2,000 developers coming from different ecosystems, where 48% of npm developers said that they release less than one breaking change per year[30]. However, a recent study[32] showed that breaking changes still occur in minor and patch updates of npm packages and that the majority of the breaking changes are type-related. These are modifications of a library that affect the presence or types of functions or other properties in the library interface, including renaming a public function, moving it to another location, or changing its type signature.

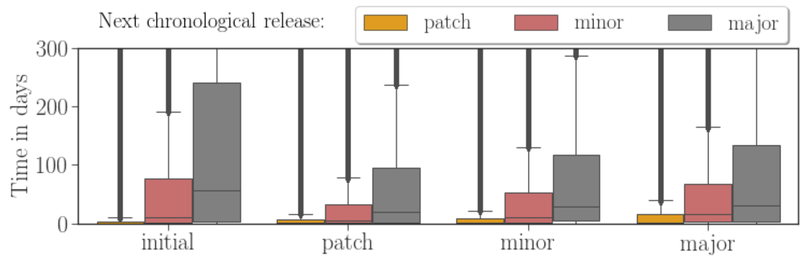


FIGURE 4 Distribution of the time until the next chronological version of npm package releases.

Summary: The way in which npm package releases use dependency constraints suggests that package maintainers are concerned about their dependency changes. We also observe, rather unsurprisingly, that the time needed to release a new package version is related to its release type (i.e., major, minor or patch).

RQ$_1$: How much technical lag is induced by direct dependencies?

This research question focuses on the evolution of the technical lag of package releases induced by their *direct* dependencies. To do so, we compute and analyse $\mathbf{deplag_{time}}(p, t)$ and $\mathbf{deplag_{version}}(p, t)$ at the release dates $t \in \mathcal{T}$ of all analyzed npm package releases p. In RQ$_4$ we will study the same but for *transitive* dependencies.

We found only 11.66M (i.e., 25.8%) outdated dependencies (i.e., package releases q required by a dependency, for which $\mathbf{techlag}_{\mathrm{time}}(q) > 0$). 7.94M (i.e., 68%) are development dependencies and 3.72M (i.e., 32%) are runtime dependencies. On the other hand, we found that 54.1% of the up-to-date dependencies are development dependencies, against 45.9% of the runtime dependencies. Given that the dataset contains 42.3% runtime dependencies and 57.7% development dependencies, the relative proportion of outdated dependencies is higher for development dependencies. This was expected, since the presence of outdated dependencies is more problematic for runtime dependencies, required to install and execute the package release, than for development dependencies, used only in a local development environment with little to no impact on the production environment.

Figure 5 visualises the evolution, on a monthly basis, of the distribution of $\mathbf{deplag_{time}}$ for all package versions released during that month, grouped by *development* and *runtime* dependencies. We notice that $\mathbf{deplag_{time}}$ tends to increase over time for both types of dependencies. To confirm our observation, we carried out a linear regression over time of the median value $\mathbf{deplag_{time}}$. We obtained a coefficient of determination of $R^2 = 0.77$ with a positive slope of 2.32, indicating a linear increase in time-based lag. This is likely because of the fact that more new releases become available over time. We also observe that the median value of the distribution never exceeded 270 days. Considering the whole observation period, we found that the median value for runtime dependencies (160 days) is lower than for development dependencies (195 days). This corroborates our previous observations about the difference between outdated development and runtime dependencies.

To compare the distribution of $\mathbf{deplag_{time}}$ for runtime dependencies to the one for development dependencies, we split the considered observation period into seven one-year periods (from 2011 until 2017)[18], each year includes all package versions that were released in it. Using the *Mann-Whitney U* test we found a statistically significant difference (p-value $< .001$) for all years except 2012 and 2016. Using *Cliff's delta* we found a small effect size that increases over time: from $d = 0$ for the first year (2011) to $d = 0.12$ for the last year (2017). In summary, development dependencies tend to be slightly more outdated than runtime dependencies, especially during the last years.

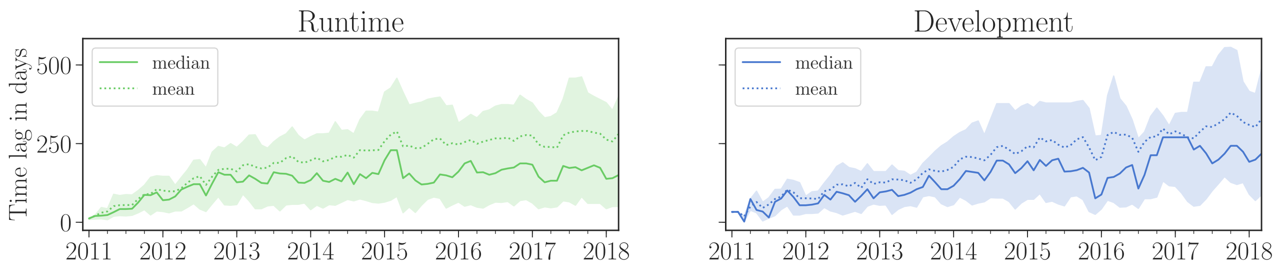


FIGURE 5 Monthly evolution of the distribution of $\mathbf{deplag_{time}}$ for all package releases, grouped by runtime and development dependencies. The shaded areas correspond to the interval between the 25th and 75th percentile.

While measuring time lag provides a good first estimate of how outdated a package release is, it is not sufficiently precise to assess the underlying source of the lag. Package releases (or package dependencies) with an identical time lag could have a different evolutionary behaviour. For example, some packages have frequent updates while others are infrequently updated; some packages may have many dependencies while others have only a few ones; some packages primarily provide patch updates while others regularly provide new major releases. Hence, comparing package releases only on the basis of time-based technical lag is not sufficient.

Because of this, we also analyse the temporal evolution of $\mathbf{deplag_{version}}$ of each package release p, computed as a function of its three version components (major, minor, patch). This version-based technical lag measurement can help to provide more insights the type of changes missed by a package release because of its outdated dependencies.

Considering all package releases over the entire observation period, we found a median $\mathbf{deplag_{version}}$ of (1,1,4) for runtime dependencies and a median $\mathbf{deplag_{version}}$ of (2,2,9) for development dependencies. Figure 6 visualises the evolution, on a monthly basis, of the distribution of $\mathbf{deplag_{version}}(\mathrm{p}, \mathrm{t}) = (\text{Major}, \text{Minor}, \text{Patch})$ for all package releases available during that month, grouped by runtime and development dependencies, and split per version component. Like for the time lag, we notice that for both types of dependencies the version lag is slightly increasing over time. We also clearly observe that version lag is higher for development dependencies than for runtime dependencies. In particular, in recent months (starting from first quarter of 2017), the version lag has increased a lot for development dependencies. This suggests that the used development dependencies (and their constraints, especially the caret) are less frequently updated than runtime ones. To confirm this hypothesis, we carried out a *Mann-Whitney U* test between the time needed before updating a version constraint within developement and runtime dependencies. We found a statistically significant difference (p-value $< .001$). Using *Cliff's delta* we found a small effect size with $d = 0.29$, indicating that the time needed before updating a version constraint is slightly higher within development dependencies.

Summary: Technical lag induced by direct dependencies in npm package releases is increasing over time due to many missed updates, including major ones. Technical lag is higher for development dependencies: 2 out of 3 outdated dependencies in npm are development dependencies.

RQ$_2$: How do constraint operators impact on technical lag?

In this research question, we explore to which extent the use of the different types of dependency constraint operators is related to the presence of technical lag. For each outdated dependency (i.e., each package release q required by a dependency, for which $\mathbf{techlag}_{\mathrm{time}}(\mathrm{q}) > 0$), we identified the operator used by the corresponding dependency constraint. Results are presented in Figure 7, showing the proportion of outdated

[18]The last 3 months in 2018 were excluded from this analysis since it does not cover a full year.

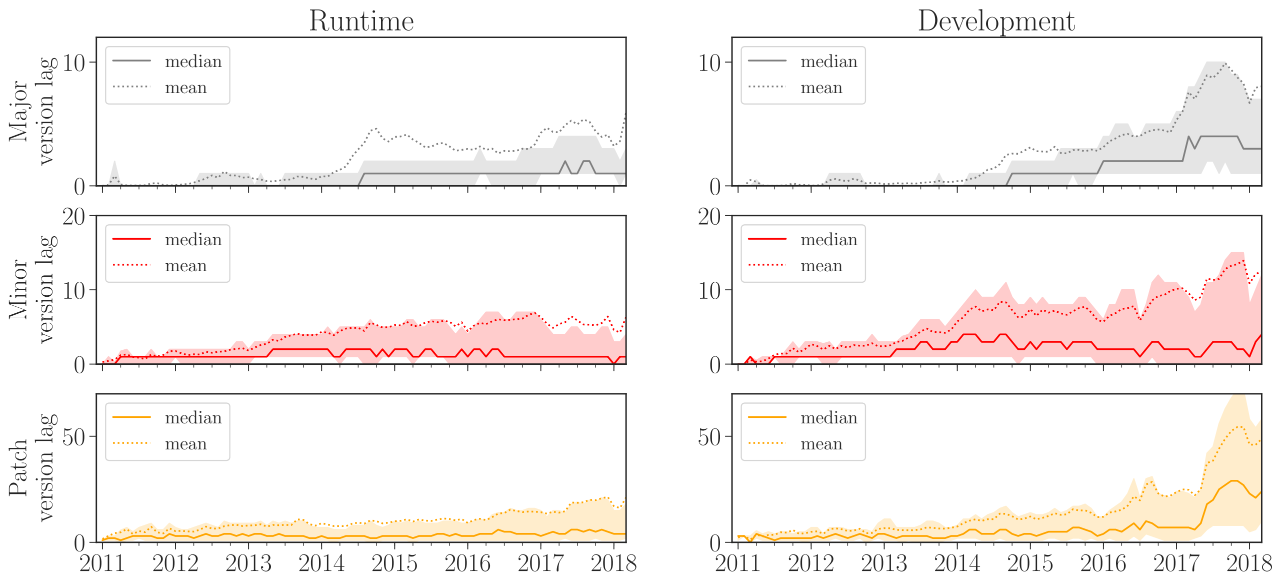


FIGURE 6 Monthly evolution of the distribution of $\mathbf{deplag_{version}}(p) = (\text{Major}, \text{Minor}, \text{Patch})$ for all package releases, grouped by runtime and development dependencies, and split per version component. The shaded areas correspond to the interval between the 25th and 75th percentile.

dependencies w.r.t. the type of constraint operator being used. We observe that the majority of outdated dependencies (15.7% and 43.5%) use the caret constraint, 26.1% (i.e., 10.6% and 15.5%) use the strict constraint and around 11.5% (i.e., 4.1% and 7.4%) of all outdated dependencies rely on the tilde constraint.

We expected such a high proportion of caret constraints because we observed in Table 2 that they account for more than 71% of all used constraints. Because of this, we also computed the proportion of outdated dependencies relative to all dependencies for each kind of constraint. We found that only 21.4% of the package releases using caret constraints are outdated, while 40% using tilde constraints and 46.5% using strict constraints are outdated. These findings confirm the hypothesis that the use of more permissive constraints (caret is more permissive than tilde, which is more permissive than strict) lowers the risk of having outdated dependencies.

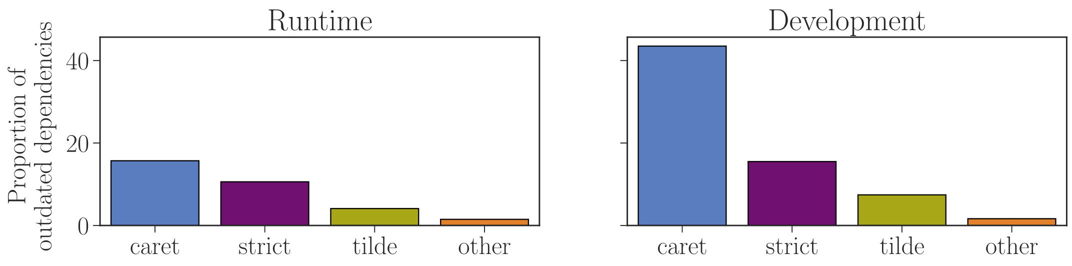


FIGURE 7 Proportion of outdated npm dependencies per constraint type, for runtime dependencies and development dependencies respectively.

To study the effect of this decision on the use of dependency constraints in npm, we analysed the evolution over time of version constraint types used by dependencies at the release date of each package release. Figure 8 shows this historical evolution of the constraint type usage, for both runtime and development npm dependencies. The *other* constraint type is gradually overtaken by the *tilde* constraint type until early 2014, suggesting that developers are becoming increasingly aware of the backward incompatible changes that might come with new released versions. Starting from February 2014, when the *caret* constraint was introduced as the default constraint in npm[19], the use of the *tilde* constraint starts to

[19] See, e.g., http://fredkschott.com/post/2014/02/npm-no-longer-defaults-to-tildes/

be replaced by the *caret* constraint. The use of the *strict* constraint type tends to remain more or less stable over time, representing about 20% of all constraint types. The use of the most permissive *latest* constraint only represents a small proportion, and is decreasing over time.

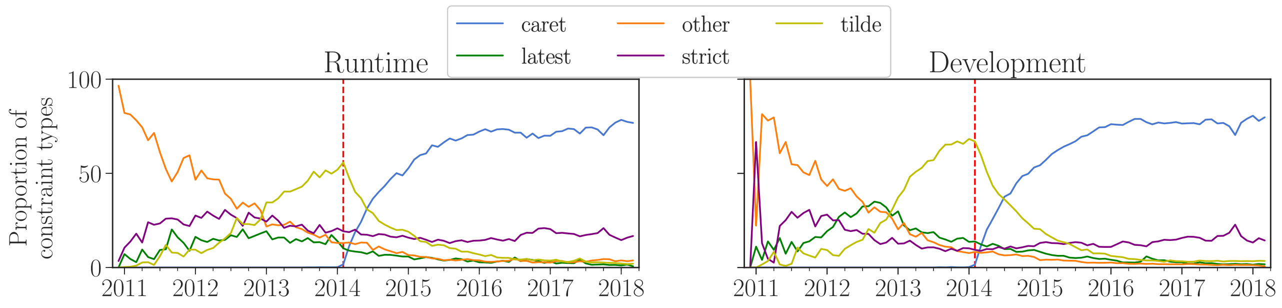


FIGURE 8 Monthly evolution of version constraint usage by **all** package dependencies.

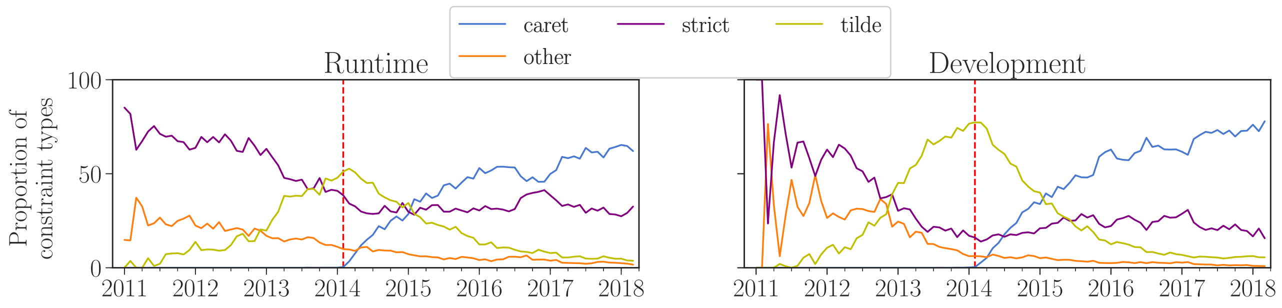


FIGURE 9 Monthly evolution of version constraint usage by **outdated** package dependencies.

We observe from Figure 8 that the tilde constraint is progressively replaced by the caret constraint, which implies that the constraint is becoming less restrictive, not more restrictive. We repeated the same analysis by only considering the evolution over time of version constraint usage by *outdated* dependencies in Figure 9. We observe that before the introduction of the caret constraint, outdated (runtime or development) dependencies were mainly used with the strict and tilde constraints, with tilde gradually taking over the usage of strict constraints until 2014. With the introduction of the caret constraint, the usage of the strict constraint stabilised over time, while caret was taking over the tilde constraint usage. This could be the reason why we observed an increasing version lag at the same time in Figure 6. To confirm this, we filtered out all dependencies using a caret constraint, and still observed an increasing major lag at the first quarter of 2014 for both runtime and development dependencies. It should not be surprising that nearly no major version lag could be observed before 2014: a limited number of package releases at that time had a version number greater than or equal to *1.0.0*. The proportion of such releases went from 20% in January 2014 to 42% in January 2015. This increase coincides with the resolution of many issues related to the use of constraints with pre-*1.0.0* version numbers[20].

Summary: npm package releases are increasingly using the caret constraint over time, Initially, the least permissive *strict* constraints were gradually being replaced by more permissive *tilde* constraints and, since 2014 these *tilde* constraints were being replaced by even more permissive *caret* constraints. Nevertheless, strict constraints continue to represent a considerable proportion of about 20% of all dependencies.

[20]https://github.com/npm/node-semver/issues/79

RQ$_3$: How does technical lag affect external applications?

The main purpose of the npm package manager is to distribute packages, to facilitate end-users to deploy them, and to allow external applications (that are not intended to be distributed through npm) to depend on them. It is interesting to measure the technical lag of such external applications depending on npm package releases, and compare their technical lag with the one of the npm package releases themselves. Since npm packages are supposed to be reused as libraries, their package maintainers are supposed to be more careful than developers of external applications that just depend on these libraries, without necessarily needing to care about other people depending on their own applications. We therefore expect to find more technical lag in external applications than in (reusable) package releases distributed via npm.

Definition 9. Technical lag framework instantiation for external applications depending on npm

To allow the npm instantiation of the technical lag framework to include external applications, we extend the set $\mathcal{P}_t$ (all npm package releases available at time t) to $\mathcal{E}_t \cup \mathcal{P}_t$, where $\mathcal{E}_t$ is the set of all external applications that are developed and distributed through GitHub[21] at time t, and that depend on at least one npm package release. All other definitions remain unchanged.

The libraries.io dataset contains references to GitHub repositories that are known to host JavaScript applications. Based on this dataset, we identified 480K GitHub repositories, focusing only on repositories that are not forks so that applications with many forks will not be considered many times in the analysis. The main purpose of forks is to propose changes to someone else's application, thus forked repositories are not meant to be considered as different applications[22]. For each of these repositories, we extracted the content of the `package.json` file for their last known commit. This file contains the list of dependencies, including the ones that target packages hosted on npm. These dependencies account for around 6.2M dependencies.

As for the previous research questions, we distinguish between runtime and development dependencies from external applications to npm package releases. We found that 53.2% of these dependencies are runtime dependencies, while 46.8% are development dependencies. 39.5% (2.5M) of these dependencies from external applications were outdated (i.e., they referred to an npm package release q for which $\mathbf{techlag}_{\text{time}}(q) > 0$). Proportionally, 48.2% of these outdated dependencies where runtime dependencies and 51.8% were development dependencies.

For all these outdated dependencies originating from external applications, we identified the operators used in their constraints. Table 3 shows the proportion of these constraint types, and compares them to the proportion observed for npm package releases. Unlike what we observed for npm package releases, where caret was the prominent dependency constraint, external applications mostly rely on strict constraints (nearly half of their runtime dependencies, 47.4%), while only around one quarter (25.2%) rely on the caret constraint. A possible explanation is that external applications are not aimed to be distributed for reuse by other packages, but mainly aim to be used in production. In this context, pinning versions of the dependencies facilitates the replication of an environment in which the application is known to work. This explains the higher use of strict constraints. In fact, replicating environments is now directly supported by npm through the "package-lock" file[23].

Dependencies	External applications				npm package releases			
	caret	tilde	strict	other	caret	tilde	strict	other
Runtime	25.2%	21.2%	47.4%	6.2%	49.2%	12.9%	33.2%	4.7%
Development	53.5%	29.3%	16.0%	1.2%	63.9%	10.9%	22.8%	2.4%
Both	39.8%	25.4%	31.1%	3.7%	59.2%	11.6%	26.1%	3.1%

TABLE 3 Proportion of constraint types used by outdated dependencies from external applications to npm package releaaes, compared to the proportion of constraint types used by outdated dependencies from npm package releases.

Let us now study the time-based technical lag $\mathbf{deplag_{time}}(p)$ of external applications $p \in \mathcal{E}_t$ depending on npm package releases at different time points $t \in \mathcal{T}$. Figure 10 shows the evolution, on a monthly basis, of the distribution of $\mathbf{deplag_{time}}$ for all external applications having their last known commit during that month, grouped by runtime and development dependencies. We observe a median value of 218 days for runtime dependencies, and 275 days for development dependencies. This is several months higher than the median values we observed for $\mathbf{deplag_{time}}$ of npm package releases in Figure 5 (160 days and 195 days for runtime and development dependencies, respectively).

We also observe that $\mathbf{deplag_{time}}$ is increasing over time for both runtime and development dependencies of external applications, especially after the first quarter of 2017. This means that external applications whose last known commit is in 2017 still had a high technical lag because

[21] GitHub is the main Open Source platform for JavaScript applications [33].
[22] https://help.github.com/articles/fork-a-repo/
[23] https://docs.npmjs.com/files/package-lock.json

of their dependencies. The figure also shows that there was no time lag for development dependencies until mid 2011. We investigated this observation and we found that all development dependencies before July 2011 were up-to-date (zero technical lag).

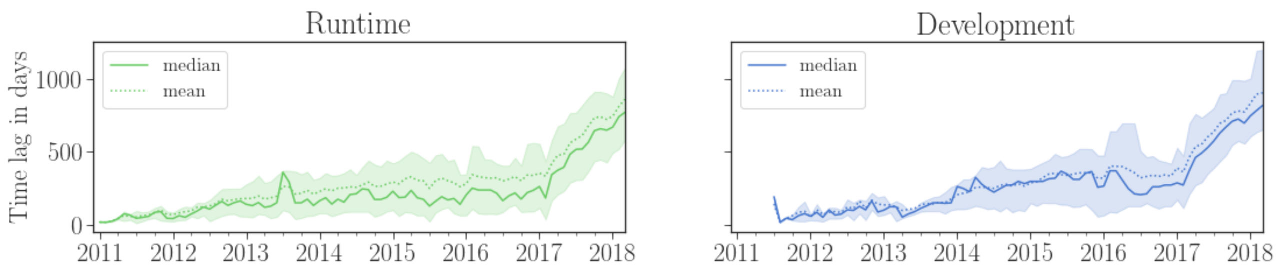


FIGURE 10 Monthly evolution of the distribution of $\mathbf{deplag_{time}}$ for all external applications, grouped by runtime and development dependencies. The shaded areas correspond to the interval between the 25th and 75th percentile.

Next, we compute the version-based technical lag $\mathbf{deplag_{version}}(\mathrm{p})$ of external applications $\mathrm{p} \in \mathcal{E}_\mathrm{t}$ at different time points $\mathrm{t} \in \mathcal{T}$. Figure 11 visualises the evolution, on a monthly basis, of the distribution of $\mathbf{deplag_{version}}(\mathrm{p}, \mathrm{t}) = (\mathrm{Major}, \mathrm{Minor}, \mathrm{Patch})$ for all external applications during that month, grouped by *runtime* and *development* dependencies, and split per version component. As was the case for the time-based lag, the version-based lag is slightly increasing over time, especially since the beginning of 2017. Focusing on the major version component, we see that it started increasing by mid 2014, just like what we observed in Figure 6. However, it is less than what we observe for minor or patch version components .

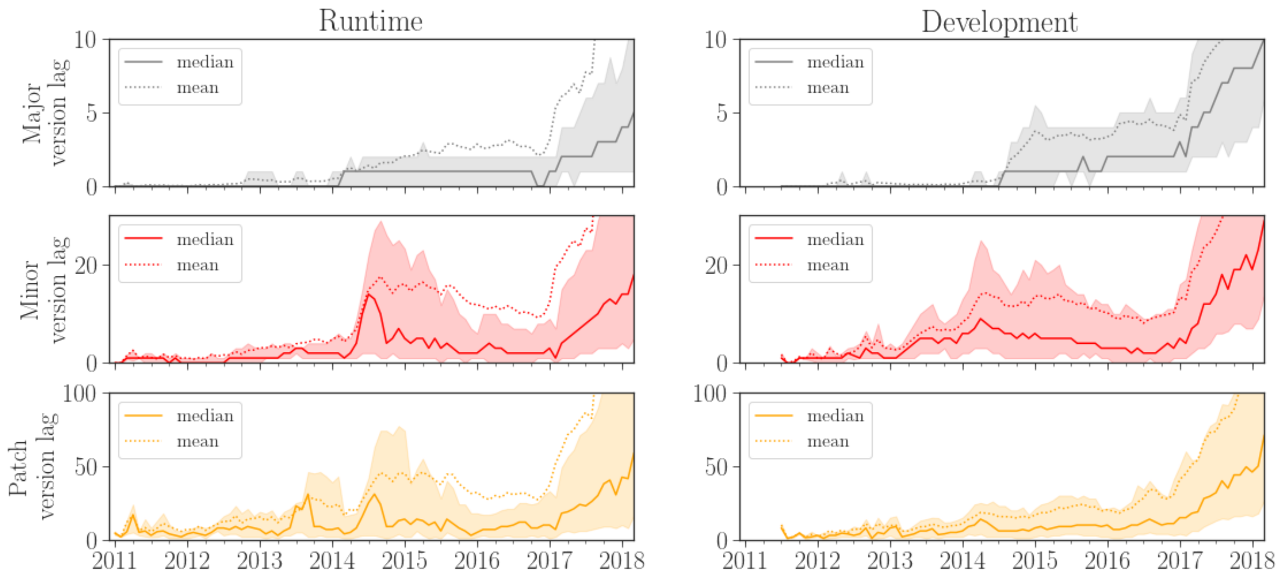


FIGURE 11 Monthly evolution of the distribution of $\mathbf{deplag_{version}}(\mathrm{p}) = (\mathrm{Major}, \mathrm{Minor}, \mathrm{Patch})$ for all external applications, grouped by runtime and development dependencies, and split per version component. The shaded areas correspond to the interval between the 25th and 75th percentile.

When we compare $\mathbf{deplag_{version}}(\mathrm{p}, \mathrm{t}) = (\mathrm{Major}, \mathrm{Minor}, \mathrm{Patch})$ between external applications ($\mathrm{p} \in \mathcal{E}_\mathrm{t}$, Figure 11) and npm package releases ($\mathrm{p} \in \mathcal{P}_\mathrm{t}$, Figure 6), we observe a difference, especially at the level of the minor and patch component. For external applications we found a median value of (1,3,9) for runtime dependencies and a median value of (2,4,10) for development dependencies (compared to (1,1,4) and (2,2,9) for $\mathbf{deplag_{version}}$ of runtime and development dependencies of npm package releases). We hypothesize that this is due to the higher proportion of strict constraints used by external applications that depend on npm packages.

Figure 12 shows the evolution of the proportion of constraint types used by external applications for their runtime dependencies to npm packages. Compared to Figure 8, we observe a much higher proportion of strict constraints, and this proportion tends to increase over time. While the proportion of the caret constraint increased since 2014, it “stabilised” at a high percentage of strict constraint, which was not the case for the npm package releases. This explains the high technical lag found for external applications relying in npm packages. We also see a tendency of a decreasing use of caret and an increasing use of strict starting in 2017. This might explain the increased technical lag that we observed since the first quarter of 2017.

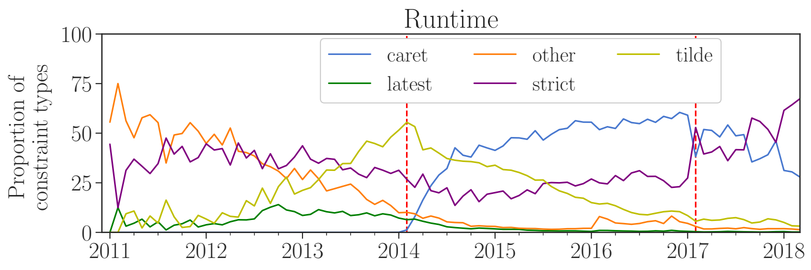


FIGURE 12 Monthly evolution of the proportion of constraint types used by runtime dependencies in external applications depending on npm packages.

Summary: The median technical lag in external GitHub applications that depend on npm packages is much higher than the technical lag found in npm package releases, and is increasing over time. This seems to be the consequence of the more frequent and increasing use of strict constraints by external applications.

RQ$_4$: How does technical lag propagate over transitive runtime dependencies?

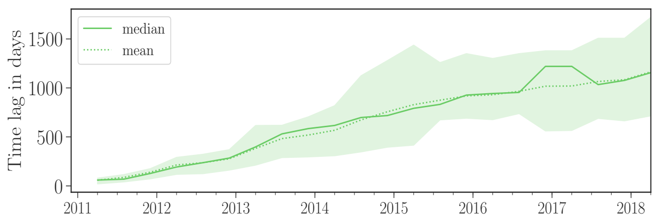
All previous research questions have focused on the technical lag induced by *direct dependencies* (i.e., $\mathbf{deplag_{time}}$ or $\mathbf{deplag_{version}}$) because these dependencies are the ones that are explicitly declared by a package maintainer, and over which they have direct control by means of dependency constraints. However, when a package release is deployed, not only its direct dependencies need to be installed but also all their dependencies, and the dependencies of these dependencies, and so on. The npm package manager installs these *transitive dependencies* in distinct subfolders, leading to a dependency graph that is a tree[24]. These transitive dependencies may induce additional technical lag on the package release being installed since the technical lag can be accumulated in the dependency tree from a level to another (deeper) level.

In this research question, we therefore quantify and analyse the *transitive* technical lag of a package release induced by all its transitive dependencies. We will do so for the time-based transitive lag $\mathbf{deplag}^{+}_{\text{time}}$ and the version-based transitive lag $\mathbf{deplag}^{+}_{\text{version}}$. While deploying a package release requires all transitive (direct and indirect) runtime dependencies to be installed, indirect development dependencies do not need to be installed during development. Indeed, to install development dependencies, only direct development dependencies and their own (transitive) runtime dependencies are required. For this reason, we restrict the analysis of RQ$_4$ to transitive runtime dependencies only.

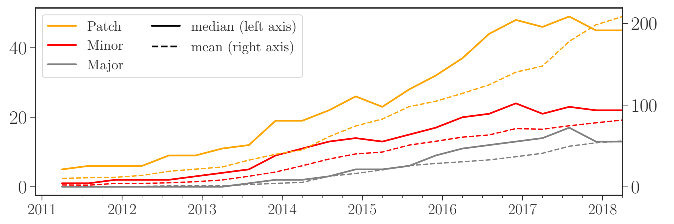
We computed the transitive runtime dependency tree of each available npm package release at the beginning of each period of 4 months (quadrimester) starting from December 2010. For each month, we considered the latest available release p of each package, and computed its time-based transitive lag $\mathbf{deplag}^{+}_{\text{time}}(p, t)$ and its version-based transitive lag $\mathbf{deplag}^{+}_{\text{version}}(p, t)$. For the whole considered observation period, we analyzed the technical lag of $672,906$ package releases of $399,522$ packages having runtime dependencies. These package releases make use of 163.8M transitive runtime dependencies on $308,243$ distinct package releases.

When analysing time-based transitive lag $\mathbf{deplag}^{+}_{\text{time}}$ we found 38.4% of all runtime dependencies to be outdated. Figure 13a shows the quadrimestral evolution of the distribution of $\mathbf{deplag}^{+}_{\text{time}}$ from December 2010 until March 2018. The graph starts in 2011 because we could not find any technical lag in December 2010 (i.e., all runtime dependencies were up-to-date). We observe that $\mathbf{deplag}^{+}_{\text{time}}$ is increasing over time. To confirm this hypothesis, we carried out a linear regression over time of the median value $\mathbf{deplag}^{+}_{\text{time}}$. We obtained a coefficient of determination

[24]see https://medium.com/learnwithrahul/understanding-npm-dependency-resolution-84a24180901b



(a) Temporal evolution of the distribution of $\mathbf{deplag}^{+}_{\mathsf{time}}$



(b) Temporal evolution of the distribution of $\mathbf{deplag}^{+}_{\mathsf{version}}$, split per version component

FIGURE 13 Quadrimestrial evolution of the distribution of transitive technical lag for runtime dependencies of all npm package releases. The shaded areas correspond to the interval between the 25th and 75th percentile.

of $R^2 = 0.95$, indicating a linear increase in transitive time-based lag. The increase is expected, since $\mathbf{deplag}_{\mathsf{time}}$ for direct dependencies was also found to be increasing over time (cf. Figure 5). Compared to the median value of time lag of $\mathbf{deplag}_{\mathsf{time}}$ for direct runtime dependencies (160 days, see Section 5.2), the median value of $\mathbf{deplag}^{+}_{\mathsf{time}}$ for transitive runtime dependencies is much higher and it exceeded 500 days in April 2013. This should not come as a surprise, given that many package releases have a deep transitive dependency tree and that lag accumulates from one level to another one.

Next, we computed $\mathbf{deplag}^{+}_{\mathsf{version}}$, the version-based lag induced by runtime transitive dependencies of all package releases. Figure 13b shows, for each version component of $\mathbf{deplag}^{+}_{\mathsf{version}}(\mathsf{p}, \mathsf{t}) = (\mathsf{Major}, \mathsf{Minor}, \mathsf{Patch})$, the evolution of the mean and median value on a quadrimestrial basis from December 2010 until March 2018. We observe that the transitive version-based lag is increasing over time.

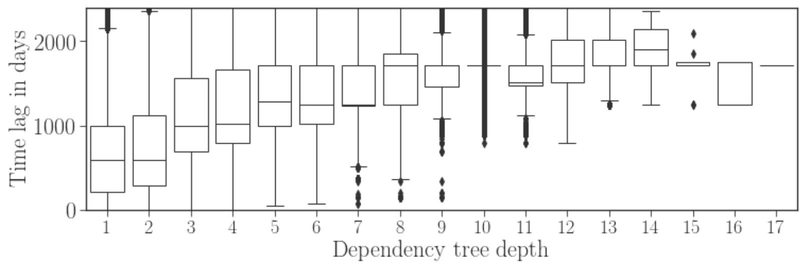


FIGURE 14 Distribution of $\mathbf{deplag}^{+}_{\mathsf{time}}$ of the latest releases of all npm packages on 13 March 2018, grouped by transitive dependency tree depth.

We also correlated the transitive time lag $\mathbf{deplag}^{+}_{\mathsf{time}}$ of package releases to their transitive runtime dependency tree depth. On the last considered snapshot (13 March 2018) we measured $\mathbf{deplag}^{+}_{\mathsf{time}}$ for the latest available package release. Figure 14 shows the distribution of these values, grouped by dependency tree depth. We observe that package releases with deeper dependency trees tend to have a higher transitive time lag. This is due to the fact that releases with deep dependency trees are more likely to have more outdated dependencies, and consequently they are more likely to have higher technical lag. Also, we observed that technical lag somehow "accumulates" from one level of the dependency tree to another one, implying that the more levels a dependency tree has, the higher the technical lag could be. To confirm this hypothesis, we calculated the correlation between the number of outdated dependencies in a dependency tree and the dependency tree depth. We found a moderately strong positive Pearson correlation ($R = 0.70$) and a strong positive Spearman correlation ($\rho = 0.91$).

Summary: Technical lag induced by transitive runtime dependencies is increasing over time and is related to the depth of the transitive dependency tree. Deeper dependency trees have a higher number of outdated dependencies, inducing a higher technical lag.

6 | DISCUSSION AND FUTURE WORK

The instantiation of the formal technical lag model to the case study of the npm package manager allowed us to empirically analyse how the technical lag of JavaScript package releases evolves over time. At the same time, the formalisation of the concept will enable comparison of this analysis with case studies of other package managers, and other types of component-based software distributions that aim to deploy systems from large collections of components.

Our findings show concrete evidence of technical lag induced by direct and indirect dependencies of both npm package releases and external GitHub applications that rely on them. This reflects the known tension between two forces that any person maintaining some software deployment faces. On the one hand, deployments would ideally use the most recent releases of their dependencies as soon as these become available, in order to benefit from the latest functionality and bug fixes. On the other hand, deployments suffer from technical lag in practice, because some dependencies do not allow for updating ("if it ain't broke, don't fix it"). Moreover, developers sometimes consciously choose not to update because they feel they do not need the new functionality provided by updates. In the specific case of security vulnerabilities, it would be acceptable to keep using an older major version of a package as long as security patches are being backported to it. In some cases, depending on the specificities of the package manager, developers don't even have a choice, as they need to rely on older releases, either because upgrading would cause co-installability conflicts[34], or because of the cost and effort required to update. Therefore, we expected to find technical lag in npm and we have quantified it in terms of aggregated time lag and version lag at package release time, thereby revealing the omnipresence of technical lag in a realistic and prominent open source case study.

We studied how version constraint operators are used in npm package dependencies, and found that it had an important effect on the technical lag of package releases. By definition, the induced technical lag is determined by the constraints imposed on the dependencies, as such constraints allow package maintainers to balance between dependency freshness and incompatibility. Only the use of the most permissive constraint operators (e.g. `latest` or `*`) do not increase technical lag. But this goes against the semantic versioning principles, as it may lead to backward incompatible updates when a new major version is released. Therefore, the use of the caret constraint (that allows for minor updates) and the less permissive tilde constraint (that only allows for patch updates) should be more frequently used for specifying dependency constraints. This was confirmed in previous work[8] where we showed that, if dependency constraints would rely on semantic versioning rules that enable automatic updates of backward compatible changes, nearly one out of five package releases would no longer suffer from technical lag. Therefore, package maintainers remain the ultimate responsible for the technical lag incurred by their package releases.

For external GitHub applications relying on npm packages, we saw an important use of strict constraints that could be explained by the need to pin dependency versions to facilitate the replication of environments, but leading to a higher technical lag. The type of dependency constraint being specified represents a package maintainer's conscious choice between "preferring to miss some updates and needing to update constraints manually" versus "preferring to benefit from dependency updates, even if that implies needing to adapt your own software to make it compatible".

Regardless of the preferred choice, it is important to offer proper measurement and testing tools to developers that indicate when, where and why technical lag increases, and whether it introduces breaking changes in the code. Such support should be part of more extensive monitoring tools (preferably part of a continuous integration process) that also take into account the presence of bugs and security vulnerabilities, unmaintained packages, known incompatibility issues, transitive dependencies, technical debt, among others. Such a tool could also provide suggestions on how to reduce technical lag by making certain constraints more permissive. Tools could as well offer support in the opposite direction to make maintainers of reusable libraries aware of the technical lag in the software that depends on them.

For the npm repository in particular, an example of a useful tool for finding and selecting reusable packages is npms.io [25]. It allows developers to search for packages with similar functionality, and select the most appropriate one among those, by relying on useful characteristics such as the package quality, maintenance or popularity. The concept of technical lag could easily be added.

An assessment of technical lag at the level of the entire package repository is also useful, in order to understand how the structure of the repository as well as the practices of the used package management tool influence technical lag. Indeed, we observed that the policies adopted by a package manager tool can push toward more or less technical lag. For example, we saw that the decision taken by npm to change the default use of tilde by the use of caret resulted in a growth of technical lag because of the increased use of a more strict constraint. A technical lag measurement tool could therefore be helpful to carry out "what if" scenarios, in order to assess upfront how particular changes in the package manager policy may affect current and future technical lag in the package repository.

From a research point of view, our technical lag framework can be used to continue exploring different ways of measuring technical lag, taking into account security vulnerabilities, closed issues, pull requests, etc. More precise lag measurements could involve a dynamic analysis of the actual source code to uncover the presence of breaking changes. This can be very challenging for a dynamically typed and interpreted language such as JavaScript, but solutions are being proposed[32]. Having obtained more detailed information about the causes of technical lag, it should become possible to provide estimation models of the effort required to reduce it.

It is also useful to compare the extent of technical lag across different software component repositories and different package managers, in order to assess which policies, practices, culture and tools lead to the best compromise. Inspired by Lauinger et al.[23], we would also like to include other types of external applications to our analysis, like deployed websites.

[25] See https://npms.io and https://docs.npmjs.com/getting-started/searching-for-packages

7 | THREATS TO VALIDITY

Our formal framework for measuring technical lag aimed to be as generic as possible, hence it should be applicable to any type of software repository as well as to other ways of measuring technical lag. The specific instantiation of this framework to the npm case study is, however, only partially generalisable, since different repositories may use different notations and ways for expressing and interpreting version numbers and version constraints, and may provide different ways to support the semantic versioning specification. As a result, the findings we have obtained for npm cannot be generalised to other package managers, since they inevitably depend on repository-specific factors such as the policies, tools, practices and values adopted by the community (cf. Bogart *et al.*[25]).

The concrete way in which we instantiated the technical lag framework for npm, and the way we have aggregated technical lag, has a direct effect on our observed findings. We do not consider this as a *real* threat to validity, since the purpose of the technical lag framework is to define and explore different useful ways of measuring technical lag, each of which may provide different results that should be interpreted differently.

Our empirical analysis of npm relied on the libraries.io dataset. There is no guarantee that this dataset is complete (e.g., there may be missing package releases), but we did not observe any missing data based on a manual inspection of the dataset. Our analysis excluded some dependencies with packages we were unable to identify on npm, but this only represented a small fraction of all dependencies (<3%) and is hence unlikely to affect the results. Considering the full set of available npm packages also constitutes a threat to validity. As explained in the data extraction in Section 5.1 we had to sanitise the dataset by excluding a number of packages before starting the analysis. Since there is no automatic way of identifying all of them, we may have missed some that should have been not considered.

The way in which we processed version numbers and dependency constraints may have influenced our results. Each dependency constraint was classified in a specific category (e.g., caret, strict, latest) depending on the syntax used by the constraint. This is, however, an oversimplification. For example, the following three constraints have the same meaning, even though they use a different notation: $\sim 1.2.0$, $1.2.x$, and 1.2. We also did not consider the difference between package releases with major version number 0 (e.g., 0.1.1) and those with a major version of 1 or above. npm treats these cases differently. For example *caret* and *tilde* have exactly the same meaning when evaluated against 0.y.z while this is not the case for higher major version numbers. A large number of npm package releases have major release number 0, and we found that 17% of the dependencies in our dataset specified a *caret* or *tilde* version constraint to a 0.y.z version. This may have affected some of our findings.

A similar threat relates to the use of prerelease tags on version numbers in npm package releases. We have excluded such releases from our analysis because they should not be treated in the same way as regular releases as they are considered to be unstable according to the npm semver "A pre-release version indicates that the release is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version".

8 | CONCLUSION

This paper focused on the important challenge that developers and maintainers face when they need to make decisions about whether and how to upgrade outdated reusable software components, during development or in production, in a certain software deployment. In many cases, those deployments are built automatically with package managers, selecting specific component releases from software component repositories, such as package management systems for the major programming languages, or Linux-based software distributions. Developers are confronted with a difficult choice. Either they stick to outdated versions of a software component, preventing them to benefit from bug and security fixes and new available functionality, or they decide to upgrade to a newer version, incurring the risk of backward incompatibilities or other technical problems. The fact that many components have explicit or transitive dependencies on many others, makes the risk assessment of upgrading components even more complex – and taking rational decisions becomes even more difficult.

We addressed this problem by providing a general formal framework for measuring how outdated (i.e., how far away from a certain "ideal" state) is a given component or its deployment. This framework is based on the notion of technical lag for a component release in a given component repository, which intends to measure the difference that deploying that release would cause with respect to some "ideal" release. We formally defined the technical lag concept through a parameterised framework that can be instantiated to a wide variety of component distributions or package managers.

The genericity of the framework makes it easy to instantiate it to deal with specific characteristics of component repositories, such as the presence of dependencies between component releases. The ability of taking into account dependencies makes the framework useful in practice, since many deployments can be modeled as a main application (or in general, a component) together with a set of direct or transitive dependencies. The framework enables to assess the effect of such dependencies on the technical lag of a component deployment, for example, by studying how outdated dependencies somewhere in the dependency tree affect the deployment. When operationalised in a tool, such technical lag measurements and assessments may help developers to explore and evaluate the effects of different component upgrading policies.

To validate the potential of the proposed technical lag measurement framework, we have instantiated it for the case of npm, one of the largest and most used package managers. We studied the technical lag of npm package releases, taking into account their direct and transitive dependencies. We considered a time-based and a version-based notion of technical lag. This required us to take into account the specificities of the npm package manager, such as how it supports the semantic versioning policy, and how it allows to specify version constraints on package dependencies. To evaluate how the framework can assist in decision making, we have used it to track the evolution of technical lag of all package releases in npm over a period of more than seven years. We also studied how different types of dependency constraints affect the overall lag, and how developers could reason about such information to decide which constraints they should use for their releases. Finally, we have identified a large collection of JavaScript applications on GitHub using npm package releases, and have studied their technical lag.

From this empirical study of npm, we found that technical lag induced by direct dependencies is increasing over time, and development dependencies tend to induce more technical lag than runtime dependencies. The main source of technical lag appears to be the type of constraints used by the dependencies. The technical lag is increasing over time because more dependencies are used with too strict constraints, in both npm packages and external applications. Due to their different purpose, the technical lag of external applications hosted on GitHub is higher than the technical lag of npm packages. We also found that the technical lag induced by transitive dependencies can be very high and is increasing over time. Moreover, the transitive technical lag is correlated with the dependency tree depth of package releases. Finally, we observed some important changes in the technical lag over time due to decisions and policy changes made in the npm package manager.

We expect that the proposed technical lag framework can be made more useful by building specific tools that developers and people deploying software and maintaining those deployments can include in their continuous integration systems. Our observed findings can be turned into actionable guidelines about npm dependency updates, and with time, be made applicable to other package managers. Our results also open the door for more research on technical lag and its measurement in reusable software libraries of other kinds, from Linux-based deployments, to Docker containers, or embedded systems built with reusable components. But we also expect that a better understanding of technical lag can be used to improve how constraints on dependencies are defined and maintained over time, so that the overall technical lag for a whole repository can be reduced. Of course, this needs the collaboration and training of a large fraction of component developers, but given the benefits we believe this is possible once i) the theoretical model is more thoroughly evaluated in other real case studies and ii) suitable tools are produced.

ACKNOWLEDGEMENTS

This research has received funding from SENECA, an European Union's H2020 Marie Sklo dowska-Curie grant, agreement No 64295. It was carried out in the context of FRQ-FNRS collaborative research project R.60.04.18.F "SECOHealth" and Excellence of Science project 30446992 "SECO-Assist" financed by FWO-Vlaanderen and F.R.S.-FNRS.

References

1. Decan Alexandre, Mens Tom, Grosjean Philippe. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering.* 2018;:1–36.

2. Krueger Charles W. Software reuse. *ACM Computing Surveys (CSUR).* 1992;24(2):131–183.

3. Kula R. G., German D. M., Ishio T., Inoue K.. Trusting a library: A study of the latency to adopt the latest Maven release. In: Int'l Conf. on Software Analysis, Evolution, and Reengineering:520-524; 2015.

4. Mostafa Shaikh, Rodriguez Rodney, Wang Xiaoyin. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis:215–225ACM; 2017.

5. Gonzalez-Barahona Jesus M, Sherwood Paul, Robles Gregorio, Izquierdo Daniel. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In: IFIP International Conference on Open Source Systems:182–192; 2017.

6. Cox J., Bouwers E., van Eekelen M., Visser J.. Measuring dependency freshness in software systems. In: International Conference on Software Engineering:109-118; 2015.

7. Zerouali Ahmed, Constantinou Eleni, Mens Tom, Robles Gregorio, González-Barahona Jesús. An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse:95–110Springer; 2018.

8. Decan Alexandre, Mens Tom, Constantinou Eleni. On the evolution of technical lag in the npm package dependency network. In: International Conference on Software Maintenance and EvolutionIEEE; 2018.

9. Mili Ali, Mili Rym, Mittermeir Roland T. A survey of software reuse libraries. *Annals of Software Engineering*. 1998;5(1):349–414.

10. Maiden Neil A, Ncube Cornelius. Acquiring COTS software selection requirements. *IEEE software*. 1998;15(2):46–56.

11. Bohner Shawn A. Extending software change impact analysis into COTS components. In: 27th annual NASA Software engineering workshop:175–182IEEE; 2002.

12. Frakes William B, Kang Kyo. Software reuse research: Status and future. *Transactions on Software Engineering*. 2005;31(7):529–536.

13. Gonzalez-Barahona Jesus M, Robles Gregorio, Michlmayr Martin, Amor Juan José, German Daniel M. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*. 2009;14(3):262–285.

14. Abate Pietro, Di Cosmo Roberto, Boender Jaap, Zacchiroli Stefano. Strong dependencies between software components. In: International Symposium on Empirical Software Engineering and Measurement:89–99IEEE Computer Society; 2009.

15. Abate Pietro, Di Cosmo Roberto, Treinen Ralf, Zacchiroli Stefano. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*. 2012;85(10):2228–2240.

16. Abate Pietro, Di Cosmo Roberto, Treinen Ralf, Zacchiroli Stefano. Learning from the future of component repositories. *Science of Computer Programming*. 2014;90:93–115.

17. Mileva Yana Momchilova, Dallmeier Valentin, Burger Martin, Zeller Andreas. Mining trends of library usage. In: International Workshop on Principles of Software Evolution:57–62ACM; 2009.

18. Raemaekers S., Deursen A., Visser J.. Semantic versioning versus breaking changes: A study of the Maven repository. In: International Conference on Source Code Analysis and Manipulation:215-224; 2014.

19. Kula Raula Gaikovina, German Daniel M., Ouni Ali, Ishio Takashi, Inoue Katsuro. Do developers update their library dependencies?. *Empirical Software Engineering*. 2017;.

20. Macho Christian, McIntosh Shane, Pinzger Martin. Automatically repairing dependency-related build breakage. In: International Conference on Software Analysis, Evolution and Reengineering:106–117IEEE; 2018.

21. Decan Alexandre, Mens Tom, Claes Maëlick. An empirical comparison of dependency issues in OSS packaging ecosystems. In: International Conference on Software Analysis, Evolution and Reengineering:2–12IEEE; 2017.

22. Abdalkareem Rabe, Nourry Olivier, Wehaibi Sultan, Mujahid Suhaib, Shihab Emad. Why do developers use trivial packages? An empirical case study on npm. In: International Symposium on Foundations of Software Engineering:385–395ACM; 2017.

23. Lauinger Tobias, Chaabane Abdelberi, Arshad Sajjad, Robertson William, Wilson Christo, Kirda Engin. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the Web. In: NDSS Symposium; 2017.

24. Decan Alexandre, Mens Tom, Constantinou Eleni. On the impact of security vulnerabilities in the npm package dependency network. In: International Conference on Mining Software Repositories; 2018.

25. Bogart Christopher, Kästner Christian, Herbsleb James, Thung Ferdian. How to break an API: Cost negotiation and community values in three software ecosystems. In: Int'l Symp. Foundations of Software Engineering:109–120ACM; 2016.

26. Constantinou Eleni, Mens Tom. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations in Systems and Software Engineering*. 2017;13(2-3):101–115.

27. Trockman Asher, Zhou Shurui, Kästner Christian, Vasilescu Bogdan. Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In: Proceedings of the 40th International Conference on Software Engineering:511–522ACM; 2018.

28. Kula Raula Gaikovina, Ouni Ali, German Daniel M, Inoue Katsuro. On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem. *arXiv preprint arXiv:1709.04638*. 2017;.

29. Nesbitt Andrew, Nickolls Benjamin. Libraries.io Open Source Repository and Dependency Metadata (Version 1.2.0) [Data set] Zenodo2018.

30. Bogart Christopher, Filippova Anna, Kästner Christian, Herbsleb James. Survey of Ecosystem Values http://breakingapis.org/survey/accessed: 28/10/2017; .

31. Bogart Christopher, Kästner Christian, Herbsleb James. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In: Automated Software Engineering Workshop:86–89IEEE; 2015.

32. Mezzetti Gianluca, Moller Anders, Torp Martin Toldam. Type Regression Testing to Detect Breaking Changes in Node. js Libraries. In: European Conference on Object-Oriented Programming (ECOOP); 2018.

33. Cosentino Valerio, Izquierdo Javier L Cánovas, Cabot Jordi. A systematic mapping study of software development with GitHub. *IEEE Access*. 2017;5:7173–7192.

34. Vouillon Jérôme, Cosmo Roberto Di. On Software Component Co-installability. *ACM Trans. Softw. Eng. Methodol.*. 2013;22(4):34:1–34:35.

□