STUDYING DEPENDENCY MAINTENANCE PRACTICES THROUGH THE MINING OF DATA FROM NPM PACKAGES

by

FILIPE ROSEIRO COGO

A thesis submitted to the Graduate Program in Computing

in conformity with the requirements for the

Degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

August 2020

Abstract

OPEN source software ecosystems have gained significant importance in the last decade. In a software ecosystem, client packages can enable a dependency to reuse the functionalities of a provider package. On the one hand, the diversity of freely reusable provider packages in those ecosystems supports a fast-paced contemporary software development. On the other hand, developers need to cope with the overhead brought by dependency maintenance. Dependencies need to be kept in an updated and working state, otherwise defects from provider packages can negatively impact client packages. Notable incidents denote the importance of timely and proper dependency maintenance. For example, in the "Equifax data breach", a vulnerability coming from an out-of-date dependency was explored to illegally obtain hundreds of millions of financial customers information Also, the "left-pad incident", in which a package with 11-lines of code was removed from npm, caused a significant downtime on major websites such as Facebook, Instagram and LinkedIn. Hence, proper dependency maintenance contributes to the viability of both individual packages and the whole ecosystem. In this thesis, we propose to leverage data from the npm ecosystem to understand the current dependency maintenance practices and provide actionable information to practitioners. Currently, npm is the largest and most popular open-source software ecosystem. We study three phenomena related to the dependency maintenance in software ecosystems: downgrade of dependencies, same-day releases, and releases deprecation. In this thesis, we discuss in detail the motivation and approach to study these three phenomena. We then perform an empirical analysis of the npm data to evaluate the driving forces behind these phenomena, as well as their prevalence and impact in the ecosystem. Based on our empirical observations, we propose a set of informed suggestions to improve dependency maintenance practices in npm.

Acknowledgments

I am extremely grateful for pursuing my Ph.D. on the Software Analysis and Intelligence (SAIL) lab under the supervision of Prof. Ahmed E. Hassan, to whom I would like to express my deepest gratitude. Thanks for your guidance and support, I am a much better researcher now than when I started this journey. I am also very fortunate to have Dr. Gustavo Oliva as my mentor during my Ph.D., I could not think of a better person. I am also thankful for building a true friendship with Gustavo during these 3 years. I am also very happy for having the chance to co-author with Prof. Cor-Paul Bezemer, from whom I learned a lot. My special thanks to all professors and staff from the School of Computing at Queen's University for always striving for excellence. I am very proud of being part of this School. For my supervisory committee, Prof. Robin Dawes and Prof. Patrick Martin, I am grateful for your guidance and advice.

I have great pride and honour to be part of the very inspiring SAIL team, where I could be contemporary of the most clever students. It was a pleasure to be with you all.

My sincere gratitude to all former SAILors that made this lab such a great place. Also, thanks to the open-source contributors of all npm packages for making this valuable data available, as well as the anonymous reviewers of my manuscripts for the criticism and the push for improvement. I am also very grateful for my colleagues and friends at UTFPR—Campo Mourão for their support and for making this PhD. a reality. Thanks to my dear friends in Canada, Guilherme Scharlack & Andressa Oliveira, Pete & Susan Rails, Mark Planeta & Emily Greer. Thanks Heng, Jonatas, and Safwat for the pleasurable walkings for a coffee and moments at the lab. Thanks my friends and relatives in Brazil, I miss you all very much. Without good friends, life makes no sense.

Last, but more importantly, I have no words to express my gratitude to my beloved family, my wife Riane, Clara, and Emanuel. The only reason for going through this are you guys! I would never get here without your support and patience, and love. Thanks mom, daddy, little and big brothers for the love and care. All this love is reciprocal.

Dedication

This thesis is dedicated to my family.

Co-authorship

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

- Filipe R. Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. 2019. An empirical study of dependency downgrades in the npm ecosystem. IEEE Transactions on Software Engineering (TSE). In Press. This work is described in Chapter 4.

- Filipe R. Cogo, Gustavo A. Oliva, Cor-Paul Bezemer, and Ahmed E. Hassan. 2020. An empirical study of same-day releases of popular packages in the npm ecosystem. Empirical Software Engineering Journal (EMSE). Under major review. This work is described in Chapter 5.

- Filipe R. Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. 2020. Deprecation of packages and releases in software ecosystems: A case study on npm. IEEE Transactions on Software Engineering (TSE). Under major review. This work is described in Chapter 6.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

SOFTWARE reuse is a central concern for Software Engineering and has been the subject of many prior studies. The effective adoption of reuse in a software project is linked with the improvement of quality, productivity, and cost reduction (Lim, 1994; Standish, 1984). As a consequence, there is a strong demand for reusable artifacts in software projects, in particular source code components (Barros-Justo et al., 2019). The principle behind source code reuse is that several software systems can reuse the same code component to perform a specific task. Despite the software reuse concept being first documented as early as 1969 (McIlroy, 1969), the design, implementation, and effective adoption of reusable source code components is an endeavour that still challenges software engineers (Gkortzis et al., 2019; Wasowski, 2020).

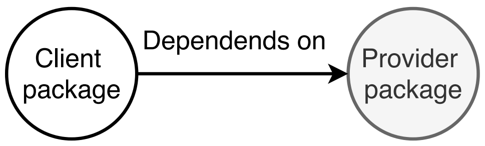


Figure 1.1: A dependency between a client and a provider package.

For this reason, researchers continue to investigate best practices regarding software reuse.

The adoption of *package managers* to distribute and reuse reusable artifacts (such as source code components and pre-compiled binaries) has become quite popular as of late. Virtually every programming language has at least one associated package manager with a centralized repository of freely reusable artifacts, maintained by several different teams as open source projects (e.g., npm is a package manager for JavaScript, pip for Python, and Maven for Java). In a package manager, the reusable artifact is bundled in the format of a *package* (other denominations include library or module), which is installed by client packages and loaded at run-time. Such package managers have emerged as a solution to leverage third-party source code reuse, particularly in the context of open source software development.

A *software ecosystem* consists of the set of interdependent packages that are deployed and co-evolve by the same package manager (Manikas and Hansen, 2013). In a software ecosystem, a *client package* reuses the code of a *provider package* (see Figure 1.1). The reuse of a provider package is enabled by a *dependency* that is specified by the client package. Dependencies can be direct (e.g., when a client package explicitly depend on a provider package) or transitive (e.g., when a client package depend on a provider package that, in turn, depends on a third package).

Currently, the largest and most popular software ecosystem[1] is npm.[2] The large number of npm packages supports the fast pace with which modern software is developed (Khomh et al., 2015; Kula et al., 2019) and contributes to the massive adoption of the npm ecosystem. To date, more than one billion package installations from npm are daily performed[3] and more than 1.3 million packages are available in the ecosystem.[4] Because of the high number of applications that rely on an npm package, several anecdotes exist to illustrate the importance of maintaining well functioning packages and a robust ecosystem. For example, in March 2016, the author of a package with 11 lines of code called left-pad removed the package from npm, causing major websites such as Facebook, Linkedin, and Instagram to stop functioning.[5] Many of these websites did not directly depend on the left-pad package, but rather had a transitive dependency on it by means of the popular react package. More recently, it was reported that a defective release of a one-line-of-code package from npm affected millions of other software projects.[6]

Dependency management is fundamental for maintaining well functioning packages and a robust ecosystem. Dependency management is the set of taken actions to promote the co-evolution of client and provider packages in a software ecosystem. A simplified dependency management cycle is depicted in Figure 1.2 and described as follows:

[1] https://insights.stackoverflow.com/survey/2020
[2] https://npmjs.com
[3] https://www.businesswire.com/news/home/20180912005283/en/npm-Registry-Crosses-Billion-Average-Daily-Downloads
[4] http://www.modulecounts.com
[5] https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/
[6] https://www.zdnet.com/article/another-one-line-npm-package-breaks-the-javascript-ecosystem/

1. Defects are discovered (such as security vulnerabilities, incompatibilities, or performance degradation) and new features are required;

2. Provider packages perform improvement changes and, as a consequence, release new versions;

3. Client packages, in turn, perform the necessary changes to adopt the updated provider version.

Nonetheless, dependency maintenance activities are performed to remedy events that can cause failures and propagate errors throughout the ecosystem (e.g., the discovery of a defect in a provider package). The grey boxes of Figure 1.2 depict the dependency maintenance activities that are the focus of this thesis:

I. Provider packages use the *deprecation* mechanism provided by package managers as a mechanism to warn client packages that the use of a certain version should be avoided (e.g., because this provider version has a serious defect, security vulnerability, or has become too obsolete);

II. Provider releases might contain errors that need to be addressed in the same day, leading provider packages to publish a follow-up *same-day release*;

III. The update of a provider package can cause malfunctioning in client packages, eventually compelling clients to *downgrade* the provider.

1.1 Thesis statement

This thesis's overall objective is to empirically study the dependency maintenance activities employed by client and provider packages in a software ecosystem. The specific

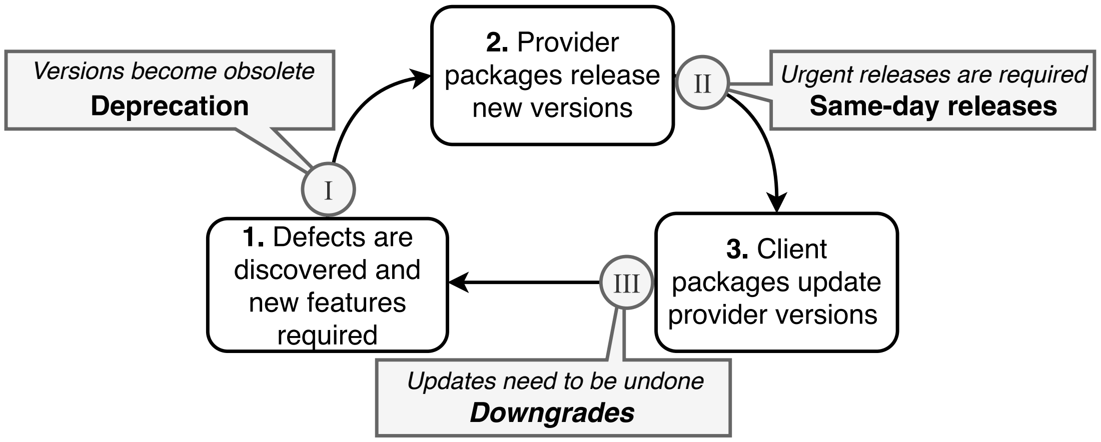

Versions become obsolete
Deprecation
2. Provider packages release new versions
Urgent releases are required
Same-day releases
I
II
III
1. Defects are discovered and new features required
3. Client packages update provider versions
Updates need to be undone
Downgrades

Figure 1.2: Dependency management cycle and the studied topics in this research proposal.

goal is to understand the current practices and to provide insights that can lead to the improvement of dependency maintenance activities, particularly those regarding the deprecation of releases, the development of same-day releases, and the occurrence of dependency downgrades. Improving current dependency maintenance practices is important not only to ensure the quality of individual packages, but also the viability of the whole ecosystem.

Empirically studying data from the npm ecosystem can provide practical insights to package developers and package manager owners on how to improve current dependency maintenance practices and ensure the viability of the whole ecosystem.

To perform our empirical studies, we collect data from the npm ecosystem. Different characteristics of npm motivate the usage of this ecosystem as a data source:

1. Dependency maintenance in npm is crucial, since packages in the ecosystem heavily depend on each other (Kikas et al., 2017). Consequently, a large amount of dependency maintenance activities are performed by client and provider packages.

2. npm implements an automatic dependency update mechanism whose principles have been adopted by other software ecosystems, while questions regarding the strengths & weaknesses of such a mechanism remain open (Bogart et al., 2016; Decan and Mens, 2019; Mezzetti et al., 2018; Decan and Mens, 2019).

3. Packages in npm are continually evolving with new releases, providing a rich source of real-world data to study modern dependency management practices.

1.2 Thesis overview

Chapter 2 of this thesis describes background information regarding dependency management in the npm ecosystem. Chapter 3 describes a literature survey regarding empirical studies on software ecosystems. Chapter 4 describes an empirical study on the downgrade of dependencies. Chapter 5 describes an empirical study of the same-day releases of popular npm packages. Chapter 6 presents our empirical study about the deprecation of packages and releases. Below, we summarize each of these three empirical studies:

1.2.1 Dependency downgrades (Chapter 4)

Provider updates can introduce failures in the client package. When such failure-inducing updates happen, client packages occasionally downgrade provider packages (Cogo et al., 2019). A downgrade is a sub-optimal yet simple workaround to guarantee the proper functioning of a package. Consequently, downgrades also contribute to the viability of the ecosystem (in contrast with the quality of an individual package). However, little is known about the downgrades of dependencies in software ecosystems. Our study about dependency downgrades has the objective of evaluating the impact of downgrades for client packages. We show that downgrades are performed because of issues that arise from the provider, but also for preventive purposes. Downgrades are also associated with a change to more conservative versioning of providers by client packages.

1.2.2 Same-day releases (Chapter 5)

Addressing issues in a timely manner is an important maintenance activity that assures the viability of the software ecosystem. Although previous papers examined same-day releases in different software distribution platforms (Lin et al., 2017; Hassan et al., 2017; Kerzazi and Adams, 2016), no prior study examined same-day releases in software ecosystems. With our study about same-day releases, we want to understand and characterize the development of such time constrained releases. We observe that, despite the short time frame within which same-day releases are developed, relevant changes are introduced in these releases. Also, same-day releases are typically adopted faster by client packages than regular releases, showing their importance to client packages.

1.2.3 Deprecation of packages and releases (Chapter 6)

Deprecation is used by provider packages to communicate to client packages that the usage of a particular release should be avoided. Prior research has exhaustively studied software deprecation at the API level (Robbes et al., 2012b; Sawant et al., 2019; Brito et al., 2016). Although many software ecosystems provide a deprecation mechanism, no prior study examined deprecation at the release level. We propose to study the release level deprecation in software ecosystems. We study the rationale behind the deprecation of packages and releases and observe that withdrawal (i.e., terminating the development of a package) is the most common rationale for deprecating all releases of a package (49%) and a defective release is the most common rationale for deprecating a specific release (63%). We also observe that a large proportion of the deprecated releases do not have a replacement release, i.e., a follow-up non-deprecated release.

1.3 Thesis contribution

The empirical results that are presented and discussed in this thesis demonstrate the value of mining data from software ecosystems, providing practitioners with actionable information to improve dependency maintenance practices. The main contributions of this thesis are:

1. We provide *client package developers* with information on the best practices for dealing with the downgrade of dependencies. In particular, we discuss that client packages should track how the adopted version of each of their providers changes over time, so debugging of troublesome provider updates becomes easier.

2. We reason about the best practices that help *provider package developers* cope with the same-day releases. In particular, we discuss the need for optimizing the quality and timing aspects of release pipelines and improving release notes.

3. We provide *package manager owners* with a set of suggestions and requirements that should be prioritized to improve the deprecation mechanism of package managers, so that the rationale behind deprecation can be better assessed.

CHAPTER 2

Dependency Management on npm

In a software ecosystem, a *dependency* is set by a *client package* to enable the reuse of the features of a *provider package*. In npm, dependencies are annotated in a configuration file called `package.json`. Each package published in npm has its own `package.json` file. An example of a hypothetical `package.json` file for a package called client_package with releases 1.0.0 and 1.0.1 can be seen on Listing 2.1. This file lists, among other pieces of information, all the published releases of a package (lines 3 and 10), a deprecation message for deprecated releases (line 8), the name of the adopted providers in each release with the associated versioning statements (lines 5, 6, 12, and 13), and the timestamp of each release (lines 18 and 19).

To set a dependency, client packages use a *versioning statement*, which determines the provider package and the respective release of that provider that is going to be

```
1   "name": "client_package",
2   "versions": {
3       "1.0.0": {
4           "dependencies": {
5               "provider_1": "2.0.0",
6               "provider_2": ">1.2.3"
7           },
8           "deprecated": "This version contains a bug."
9       },
10      "1.0.1": {
11          "dependencies": {
12              "provider_1": "2.0.1",
13              "provider_2": ">1.3.0"
14          }
15      }
16  },
17  "time": {
18    "1.0.0": "2016-11-24T00:48:15",
19    "1.0.1": "2017-02-08T13:26:38"
20  }
```

Listing 2.1: A hypothetical *package.json* file

adopted by the client package. For example, in Listing 2.1 (line 5), the client package client_package on version 1.0.0 depends on a provider package provider_1. In this dependency, the versioning statement `"provider_1":"2.0.0"` is used. As a result, whenever the package client_package is installed, the version 2.0.0 of package provider_1 is also be installed (and eventually loaded at run-time) as a dependency of client_package. When a package is installed from npm, the provider packages that are used by means of *transitive dependencies* are also installed (and eventually loaded at run-time). An example of transitive dependency is a client package c that depends on package p_1 that, in turn, depends on package p_2. In this example, c directly depends on p_1 and transitively depends on p_2. Therefore, when c is installed from npm, p_1 and p_2 are also installed.

A versioning statement can be one of two types: a specific version (e.g., `"p":"1.2.3"`) or a version range (e.g., `"p":">1.2.3"`). The *specific version* statement is satisfied by

Table 2.1: Operators in the grammar of npm version range.

Operator(s)	Definition	Example
>, <, >=, <=	Allows, respectively, any version greater, smaller, greater or equal, or smaller or equal to a given semantic version number.	`"p":">1.0.0"` is satisfied by any version of *p* greater than 1.0.0 (e.g., 1.0.2, 1.2.0, or 2.0.0).
~ (tilde)	Allows changes to the least precedent (left-most) level of the semantic version number. Intuitively, the tilde operator resolves towards a patch update of the provider.	`"p":"~1.2.3"` is satisfied by any version of *p* greater than or equal to 1.2.3 and less than 1.3.0. Still, `"p":"~1.2"` is satisfied by any version of *p* greater than or equal to 1.2.0 and less than 1.3.0.
∧ (caret)	Allows changes that do not modify the non-zero least precedent level in a semantic version number. Intuitively, the caret operator resolves towards a minor update of the provider.	`"p": "∧1.2.3"` is satisfied by any version of *p* greater than or equal to 1.2.3 and less than 2.0.0. Still, `"p":"∧0.2.3"` is satisfied by any version of *p* greater than or equal to 0.2.3 and less than 0.3.0.
— (hyphen)	Allows an inclusive set of versions.	`"p": "1.2.3—2.3.4"` is satisfied by any version greater than or equal to 1.2.3 and less than or equal to 2.3.4.
omit a semantic version level or replace it by `"x"`	Allows changes in the omitted/replaced semantic version level.	Both `"p":"1.x"` or `"p":"1"` are satisfied by any version of *p* greater than or equal to 1.0.0 and less than 2.0.0. Still, both `"p":"1.2"` or `"p":"1.2.x"` are satisfied by any version greater than or equal to 1.2.0 and less than 1.3.0.
`"*"`, `"latest"`, `"last"`, `""`	Resolves to the largest version available.	Both `"p":"*"` or `"p":""` are satisfied by the largest version of *p*.
||	Combines two or more versioning statements in a logic 'OR'.	`"p":"∧2.0.0 || ~3.0.0"` indicates that any of the statements `"∧2.0.0"` or `"~3.0.0"` are satisfied by any version of *p* in accordance with the statement definition.

a unique version of a provider, defined by the right-hand side of the versioning statement (i.e., version 1.2.3). The *version range* statement is satisfied by a range of versions of a provider (i.e., any version greater than 1.2.3). A version range statement has three parts: the provider to which it refers (`"p"`, in the previous example), an *operator* (`">"`, in the previous example), and a numerical part (`"1.2.3"`, in the previous example). The combinations of operator and numerical part define the range of provider versions that can be satisfied by the versioning statement. In fact, there is a grammar for defining a version range in npm. Such a grammar relies on a set of operator whose definitions and examples are provided on Table 2.1.[1]

[1]The grammar in Backus-Naur form can be found at `https://www.npmjs.com/package/semver`.

The *resolved version* is the actual version of the provider package that is going to be loaded as a dependency by the client package. When a version range is used, the resolved version corresponds to the largest provider version that satisfies the range. Therefore, version range statements are used when client packages wish to perform an *implicit update* of the provider without having to change their versioning statement. When a provider releases a new version that is satisfied by the existing version range statement by a client package, this provider is implicitly updated. For example, in Listing 2.1 (line 6), the version range specified by client_package in version 1.0.0 will lead to an implicit update whenever provider_2 releases a new version that is larger than 1.2.3. An implicit update contrasts with an *explicit update*, in the sense that the latter requires a modification of the versioning statement by the client package so that the resolved provider release is updated. For example, if a client package c uses a versioning statement `"p":"<1.2.3"` and p releases version 1.2.4, then an update will only be performed after the modification of the versioning statement to `"p:">=1.2.4"` (or any version range statement that satisfies the release 1.2.4 of p). Client packages can also perform a *downgrade* of the resolved provider release by restricting the versioning statement to an older provider release.

The release numbering of an npm package follows the *semantic version* specification.[2] According to this specification, a version number of a release is comprised of three levels, namely: major, minor, and patch. For instance, in release 1.2.3, the number 1 stands for the major level, the number 2 stands for the minor level, and the number 3 stands for the patch level. The semantic version also specifies simple change-related rules for developers to determine how one of the three levels should be incremented when a release is published. In summary, a *major release* should be published

[2] https://semver.org

whenever a backward-incompatible change is introduced (e.g., an API change). A major release must yield the increment of the major version level, for example, from 1.2.3 to 2.0.0. A *minor release* should be published when some new backward-compatible change is introduced. A minor release must yield the increment of the minor level of the version number (e.g., from 1.2.3 to 1.3.0). Finally, a *patch release* should be published when a bug fix is introduced. A patch release must yield the increment of the patch level of the version number, such as from 1.2.3 to 1.2.4. Although the adoption of the semantic version specification is not mandatory, a prior study shows that, in general, packages in npm comply with this specification (Decan and Mens, 2019). The mechanism to resolve a provider version relies on the precedence between version numbers, since npm needs to know if a particular version number is greater than, less than, or equal to another version number. Similarly to decimal numbers, semantic version numbers are compared initially by the magnitude of their major level, then by their minor and patch levels. For example, version 3.2.1 is lower than versions 4.0.0 (by a major), 3.3.1 (by a minor), and 3.2.2 (by a patch), but greater than versions 2.2.1 (by a major), 3.1.1 (by a minor), and 3.2.0 (by a patch).

A client package can set a provider package as either a development or a production dependency. A provider package that is set as a development dependency (so-called *development provider*) is loaded only at the development environment (e.g., the source code repository to which developers commit changes). Consequently, development providers are not loaded when the client package is installed from npm. For instance, test frameworks are generally development providers, since they need to be loaded by the client package developers but not by the client users. As a consequence, issues that arise from development providers do not affect the deployed client package (i.e., in the production environment), making the reaction to such issues less urgent. In turn, provider packages that are set as production dependencies (so-called *production provider*) are loaded both at the production and development environments. When a client package is installed from `npm`, providers that are set as production dependencies are also installed with their respective resolved versions and are loaded at runtime. Production and development dependencies are listed separately in the `package.json` file as `dependencies` and `devDependencies`, respectively.

CHAPTER 3

Literature Survey

THIS thesis aims to understand and support dependency maintenance practices. Therefore, this chapter presents related literature with dependency maintenance in software ecosystems. In particular, the chapter discusses our literature selection process (Section 3.1) and then surveys related studies on npm (Section 3.2) and other software ecosystems (Section 3.3).

3.1 Literature selection

Software ecosystems have gained attention from the software engineering research community in the last decade, driven by the popularity of the package managers that host such ecosystems. Our objective is to empirically study the current dependency

Table 3.1: Venues from which a initial list of ten years old papers was obtained for the literature review.

Venue type	Venue name	Abbreviation
Journal	IEEE Transactions on Software Engineering	TSE
Journal	IEEE Software	IEE Software
Journal	Empirical Software Engineering	EMSE
Journal	Journal of Systems and Software	JSS
Journal	ACM Transactions on Software Engineering and Methodology	TOSEM
Conference	International Conference on Software Engineering	ICSE
Conference	ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering	ESEC/FSE
Conference	International Conference on Automated Software Engineering	ASE
Conference	International Conference on Software Maintenance and Evolution	ICSME
Conference	International Conference on Software Analysis, Evolution, and Reengineering	SANER
Conference	International Conference on Mining Software Repositories	MSR

maintenance practices of packages in software ecosystems (more specifically, ecosystems that are set around packaging platforms for a programming language). To select the related literature, we search for empirical studies about *software ecosystems*, *dependency management*, and *dependency maintenance* that were published as full papers in the last ten years in the major software engineering venues. Table 3.1 lists the venues from which papers were searched. To filter out irrelevant papers, we read the abstract and remove papers that do not relate to our interest topics. From the initial list of selected papers, we perform a forward snowballing process (Wohlin, 2014) (i.e., following each paper's reference list) to include other relevant literature that is not included in the initial list. Papers that do not directly relate to software ecosystems are only selected if they provide relevant background on dependency maintenance.

3.2 Mining data from the npm ecosystem

In this section, we survey the empirical studies that use data from the npm ecosystem. We group related papers by their studied topic.

Dependency updates: Prior studies examine the risks and causes of failed updates in the npm ecosystem. Decan et al. (2019) studied the dependency network of seven software ecosystems (including npm) and concluded that developers can face issues when updating providers. Because packages in those ecosystems heavily depend on each other, the authors highlight the fact that a defect in one package can affect many others. In a survey with developers from 18 ecosystems, Bogart et al. (2016) show that provider packages in npm often introduce changes that require modifications in the client code, which can impact the rate at which providers are updated. Mirhosseini and Parnin (2017) study whether the usage of automated dependency management tools in npm can encourage client packages to update their providers. The authors show that, on average, client packages that use such automated tools update their providers 1.6 times as often as client packages that do not use such tooling. Decan et al. (2018) shows that, in average, updates in npm take from 12 to 22 days, depending on whether it is a major, minor, or patch update. Abdalkareem et al. (2017) surveyed 88 npm developers to understand the usage of "trivial packages" (packages that implement simple and trivial tasks) in this ecosystem. Respondents of the survey (56%) point the overhead caused by the need for updating as one of the drawbacks of using trivial npm packages. Zerouali et al. (2019b) analyzed the technical lag induced by direct dependencies in npm over seven years period. Technical lag is defined by as the extent to which the adopted provider version by client packages are lagging behind the latest provider version. The authors found that the median technical lag is 1 major, 1 minor, and 4 patch releases.

Such technical lag should be mitigated whether client packages perform the required updates on their provider packages. Also, Zerouali et al. (2019c) analyzed the technical lag of outdated installed packages in Docker containers and found that the technical lag on such containers is one to two versions.

Summary: Client packages in npm evolve not only by modifying their codebase but also by updating their providers. However, the update of a provider version by a client package needs to consider the tradeoff between the benefit of accessing the latest provider features and the risk of defect-introducing changes in the new provider version.

Semantic version adoption & backward-incompatible releases: Wittern et al. (2016) observed that from 2011 to 2015, the proportion of client package that adopt at least one provider package increased from 23% to 81% and that the semantic version specification is not always followed by provider packages when they increment a version number. The authors highlight that the lack of compliance with the semantic version specification can be problematic, since backward-incompatible changes could be introduced when minor or patch levels of the provider version are incremented. As a consequence, these backward-incompatibilities would manifest themselves unexpectedly on the client package side. In contrast, Decan and Mens (2019) observed that versioning statements used by client packages generally accept implicit updates of providers, suggesting that, although client packages are vulnerable to backward-compatible changes, such changes are usually not introduced by provider packages. Mezzetti et al. (2018) propose a technique called type regression that can detect backward-incompatible changes in the provider package that cause a failure in a client package.

However, this technique is only accurate when providers are used by a large number of client packages. Mujahid et al. (2020) describes a technique that leverages the test suites of client packages that adopt the same set of providers to early discover backward-incompatible releases of providers.

Summary: A reason that is commonly associated with failed updates is the introduction of backward-incompatible changes in minor or patch releases—a scenario for which developers disregard the semantic version specification. However, an intrinsic agreement between packages must exist to avoid such a scenario, otherwise the npm ecosystem would become too unstable.

Dependency vulnerabilities: Zerouali et al. (2019a) identified that vulnerabilities are common in official Docker containers using npm packages as dependencies. Kikas et al. (2017) also performed a study over the dependency network of the npm, RubyGems, and Cargo[1] (for the Rust language) ecosystems. The authors identified a set of key packages that, if vulnerable, could negatively impact over 30% of the other packages in the ecosystems. Hejderup et al. (2018) argue that tools to alert client packages regarding the usage of vulnerable providers can be improved by considering the function call graph of client and provider packages. Vulnerabilities in npm were also studied by Zapata et al. (2018). The authors also argue that a function-call analysis is needed to improve current vulnerability checking tools. Pfretzschner and ben Othmane (2017) catalogue four attacks that can explore dependencies in npm and how to avoid them. Zimmermann et al. (2019) study vulnerability issues that are introduced by npm dependencies and provide mitigation strategies. The authors highlight that the lack of

[1]https://crates.io

dependency maintenance can cause client packages to depended on vulnerable code of providers.

Summary: Dependency maintenance is an important factor to avoid depending on vulnerable code. Current dependency vulnerability tools should also consider dependencies at the function level to improve accuracy.

3.3 Mining data from other software ecosystems

Prior empirical studies focused on the management of dependencies versioning in different software ecosystems. In the scope of the Android ecosystem, McDonnell et al. (2013) analyzed the relation between the adoption of the versions an API and the stability of the API. The authors show that client packages avoid to update towards unstable APIs (i.e., APIs that change frequently). Ruiz et al. (2016) analyzed the rationale for Android applications (client packages) to update their ad libraries (provider packages). In particular, they found that fixing a bug and improving performance are some of the reasons to update an ad library. Derr et al. (2017) performed a survey with Android developers to understand how they update provider packages. They observed that preventing incompatibilities is the second-ranked reason as to why developers prefer to use outdated providers. Li et al. (2018) analyzed API deprecation in the Android platform.

The management of dependencies versioning was also studied in ecosystems for packages written in Java. Bavota et al. (2015) identified bug fix as a factor that influences the update of provider packages. Also, the authors observed that packages that have common dependencies or common developers are more likely to be updated.

Kula et al. (2015) observe that client packages are becoming more inclined in adopting the latest version of a provider package. Kula et al. (2017b) show that vulnerability and security advisories play a role in the decision of updating a provider version. Sawant et al. (2018c), Zhou and Walker (2016), Brito et al. (2016), and Ko et al. (2014) analyzed the phenomenon of API deprecation on ecosystems for the Java language. Cox et al. (2015) describe a set of metrics to evaluate the "dependency freshness" (the difference between the resolved version and the latest version of a provider) of Java packages used by an industrial software system. Digkas et al. (2018) performs a longitudinal analysis on how technical debt is addressed by 75 randomly selected Java packages available in the Apache Software Foundation Index.[2]

Robbes et al. (2012b) studied how changes in the API of a provider package propagate back to the client package in the Pharo[3] ecosystem (for the Smalltalk language). Changing the API signature of a provider is one of the actions that cause backward-incompatible changes. Normally, the client package cannot avoid such changes when they are introduced in a minor or patch release of the provider. Constantinou and Mens (2017) study the social-technical evolution of packages from the Ruby ecosystem. The authors argue that contributors abandoning has a significant impact on the evolution of packages in this ecosystem. Finally, Manikas (2016) perform a systematic literature review and Manikas and Hansen (2013) a systematic literature mapping on software ecosystems.

[2]https://projects.apache.org/projects.html?language#Java
[3]http://catalog.pharo.org

Summary: Prior empirical studies demonstrate that software ecosystem dependencies are a rich source of data. When properly analyzed, dependency data can provide practitioners with actionable insights regarding the best practices for dependency maintenance. Nonetheless, none of the prior studies perform an in-depth examination of specific dependency maintenance activities, namely the downgrade of dependencies, the publishing of same-day releases, and the deprecation of packages and releases.

In Sections 4.6, 5.2, and 6.6, we discuss additional related work on dependency downgrades, same-day releases, and release deprecation. We also compare the findings obtained in these related work with the findings of our empirical studies.

CHAPTER 4

An Empirical Study of Dependency Downgrades

A downgrade indicates that the adopted version of a provider package is not suitable for the client package at a certain moment. In this chapter, we conducted an empirical study of the dependency downgrades in the npm *ecosystem. Our study indicates that there are two categories of downgrades according to their rationale: reactive and preventive. The reasons behind reactive downgrades are defects in a specific version of a provider, unexpected feature changes in a provider, and incompatibilities. In turn, preventive downgrades are an attempt to avoid issues in future releases. We also investigated how the versioning of dependencies is modified when a downgrade occurs. We observed that client packages have the tendency to become more conservative regarding the update of their providers after a downgrade. Finally, we observed that downgrades that follow an explicit update of a provider package occur faster than downgrades that follow an implicit update. Explicit updates occur when the provider is updated by means of an explicit change to the versioning specification (i.e., the string used by client packages to define the provider version that they are willing to adopt). We conjecture that, due to the controlled nature of explicit updates, it is easier for client packages to identify the provider that is associated with the problem that motivated the downgrade.*

An earlier version of this chapter is published in the IEEE Transactions on Software Engineering journal (TSE) (Cogo et al., 2019).

In this chapter, we describe our study about dependency downgrades. The chapter is organized as follows. Section 4.1 provides an introduction to our study and a summary of our main findings. Section 4.2 describes our approach to detect downgrades. Section 4.3 explains how we collected and processed the data from npm. Section 4.4 presents the motivation, approach, and findings to the aforementioned RQs. Section 4.5 presents a discussion about our findings. Section 4.6 presents the related work and Section 4.7 presents the threats to the validity. Finally, Section 4.8 concludes our study.

4.1 Introduction

Prior research shows that code reuse is related to the improvement of developers' productivity, software quality, and time-to-market of software products (Lim, 1994; Abdalkareem et al., 2017). In the last decade, software ecosystems arose as an important mechanism to promote and support code reuse, in particular ecosystems that are set around packaging platforms for a programming language (Manikas and Hansen, 2013; Serebrenik and Mens, 2015). Such platforms are built upon the notion of *dependencies* between packages. A dependency relationship enables a *client* package to reuse a certain version of a *provider* package.

In several ecosystems, client packages can specify a dependency as either a dependency to a *specific version* or a *range of versions* of a provider. If a range of versions is specified, then the provider is *implicitly updated* whenever a new version satisfying this range is released. If a specific version is used, an update can only happen by switching it to a newer specific version or by using an appropriate version range. In addition, packages in software ecosystems typically adopt a version numbering scheme to communicate changes that are introduced in new releases. A popular scheme is the Semantic Version specification, in which a version number comprises three digits separated by a dot (e.g., 1.0.0). The first digit represents the major level of the version number, commonly incremented whenever a backward-incompatible API change is introduced in a new release. The second one represents the minor level, commonly incremented whenever a new backward-compatible feature is introduced, and the third one represents the patch level, commonly incremented whenever a bug fix is introduced.

The benefits and drawbacks of updating providers in software ecosystems have been extensively studied (Bavota et al., 2013; McDonnell et al., 2013; Wittern et al., 2016; Derr et al., 2017; Kula et al., 2017a,b; Ihara et al., 2017). On the one hand, updating providers enables a client package to benefit from bug fixes, new functionalities, security enhancements, and novel APIs. On the other hand, updating providers also makes a client package more susceptible to potential problems in the new version of the provider. In the latter case, client packages might end up *downgrading* a provider, i.e., reverting it to an older version.

In this chapter, we study *why* and *how* downgrades occur. Our motivation is based on the following observations:

- *Downgrades naturally indicate that one or more provider versions caused problems to the client package.* Despite being a natural indication of issues, downgrades can be a simple and rapid workaround for specific issues that arise when a provider package is updated. While prior research indicates that backward-incompatible changes (a.k.a., breaking changes) (Salza et al., 2018), bugs (Mirhosseini and Parnin, 2017; Mileva et al., 2009), and vulnerabilities in the provider package (Decan et al., 2018) motivate downgrades, the actual reasons behind downgrades have not been thoroughly investigated. Our objective is to study the reasons that motivate downgrades, such that practitioners can be aware of the typical cases that lead to this type of workaround.

- *Many software ecosystems platforms allow client packages to accept implicit updates of provider packages.* Such automatic updates can hinder the identification of the provider package behind a certain issue. As a consequence, the downgrade of such a provider might be delayed, or even the downgrade of unrelated providers might occur. Moreover, we hypothesize that such an automated update mechanism may lead to problematic updates which, in turn, might force client packages to abandon automated updating altogether.

- *Downgrades increase the technical lag of client packages.* The technical lag represents the extent to which client packages are missing the latest features and bug fixes from a provider. Measuring the technical lag that is introduced when a downgrade is performed should help practitioners understand the side-effects of this workaround.

All of the aforementioned issues raise important questions about how to reduce the potentially adverse side-effects of a downgrade. Elucidating detailed reasons behind downgrades, as well as how client packages perform these downgrades, would provide a more in-depth understanding of downgrades and their consequences, ultimately fostering further research and tool development to support package developers.

To conduct our study, we collected data from npm,[1] the largest ecosystem supporting the Javascript programming language, containing more than 600K reusable packages.[2] According to a survey by StackOverflow[3], JavaScript was the most commonly used programming language in 2018, with 69.8% out of more than 100K respondents affirming their use of JavaScript. In addition, managing dependencies in npm is a growing business and a number of commercial tools to aid in this task are available (Bogart et al., 2016; Mirhosseini and Parnin, 2017), such as dependency bots are deployed on package's codebase to monitor dependency updates.[4] In our study about dependency downgrades, we address the following research questions:

RQ1. Why do downgrades occur? We observed two types of downgrades: *reactive* and *preventive*. The main reasons behind reactive downgrades are: defects in the provider version (during build-time, run-time, or development-time), unexpected feature changes in the provider, and incompatibilities (between provider versions, or with Node version). Preventive downgrades occur by pinning a provider package to a prior version in an attempt to avoid issues in future releases of this particular provider. Preventive downgrades can be triggered by recommendations from automated tools.

RQ2. How is the versioning of providers modified in a downgrade? Downgrades are commonly performed by choosing a specific old version of the provider (62%) instead of specifying a range of acceptable old versions (38%). In 75.5% of the client releases containing a downgrade, only a single provider is downgraded. In 48% of

[1] `https://npmjs.com`
[2] On the time at which the study was performed (January–April 2019)
[3] `https://insights.stackoverflow.com/survey/2018/`
[4] `https://dependabot.com`

the downgrades, the provider version is reduced by a minor level (e.g., from 2.1.0 to 2.0.0). In addition, we calculated the technical lag induced by downgrades, i.e., the number of releases that are back skipped when a provider is downgraded. We observed that downgrades of major version levels (e.g., from 2.0.0 to 1.2.3) introduce more technical lag than downgrades of minor and patch version levels.

RQ3. How fast do downgrades occur? Half of the downgrades are performed at a rate that is 2.6 times as fast as the typical time-between-releases of their associated client packages. The median time to downgrade an implicitly updated provider is roughly 9 times higher than that for an explicitly updated provider. In specific, only 5.6% of the downgrades are performed in within 24 hours after the update of the provider.

Our key contribution is providing empirically-sound evidence from cross-linked data regarding why and how downgrades occur on npm, while also discussing the implications of our findings to client package developers. As an additional contribution, we provide an algorithm to recover a branch-based ordering of releases, which may be reused by other researchers studying downgrades (and updates) on npm. Finally, we provide a supplementary material with the data that is used in this study [5] as a means to bootstrap other studies in the area.

4.2 Downgrade detection

One of the requirements for a large-scale study about dependency downgrades is to automatically detect when a client package perform a downgrade of a provider package.

[5] https://github.com/SAILResearch/replication-npm_downgrades

A downgrade is detected whenever the resolved provider version decreases between two adjacent client releases. More formally, a downgrade is defined as an event that occurs between two adjacent releases $\langle r_{C,i-1}, r_{C,i}\rangle$ of a client package C. When the resolved version of a provider package P in $r_{c,i}$ is smaller than the resolved version of this same provider in $r_{C,i-1}$, we say that P was downgraded by C in $r_{c,i}$. Hence, given a list of R_C releases of the client package C with the respective version numbers and timestamps, the definition of a downgrade of a provider P by the client C depends on the ordering of $r_{C,i-1}$ and $r_{C,i}$, $\forall i \in \{1 \ldots |R_C|\}$. To detect downgrades, we sort the releases of the client packages according to a branch-based ordering algorithm, which we motivate and describe below.

4.2.1 The problem with chronological versus numerical ordering of client versions

Prior studies use either a chronological (Ihara et al., 2017; Salza et al., 2018) or a numerical ordering (Kikas et al., 2017; Wittern et al., 2016) to recover the release history of packages. However, assuming those orderings leads to inconsistencies in how the resolved provider version changes from one client release to another. Below, we show that none of these orderings are suitable to detect updates and downgrades in the resolved provider packages.

When analyzing the order of package releases in npm, we observed that several releases can be actually maintained in parallel. Releases are developed in parallel because, even with the existence of releases with a higher numerical order, a release with a lower numerical order might need to be patched. For example, in Figure 4.1, even though the release 2.0.0 was already available, the release 1.1.2 had to be published in order to patch the release 1.1.1. Hence, the version 1.1.1 is considered adjacent to both versions 1.1.2 and 2.0.0. Because the numerical and chronological ordering are linear, they are not suitable to represent the parallel releases of npm.

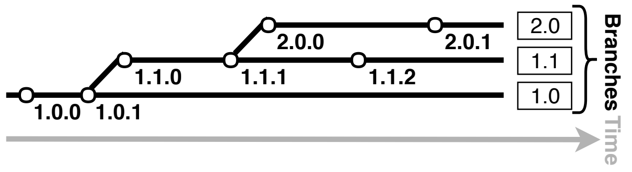


Figure 4.1: Development of parallel versions in npm.

Applying the chronological and numerical orderings to the releases that are shown in Figure 4.1 would yield the following results ($\prec$ denotes a precedence relation):

Chronological:

1.0.0 $\prec$ 1.0.1 $\prec$ 1.1.0 $\prec$ 1.1.1 $\prec$ 2.0.0 $\prec$ 1.1.2 $\prec$ 2.0.1

Numerical:

1.0.0 $\prec$ 1.0.1 $\prec$ 1.1.0 $\prec$ 1.1.1 $\prec$ 1.1.2 $\prec$ 2.0.0 $\prec$ 2.0.1

Branch-based:

1.0.0 $\prec$ 1.0.1 $\prec$ 1.1.0 $\prec$ 1.1.1 $\prec$ 1.1.2

$\curlyvee$ 2.0.0 $\prec$ 2.0.1

For a client package that releases according to Figure 4.1, analyzing the changes in the resolved provider version between adjacent client releases would produce different results, depending on the assumed ordering. Figure 4.2 illustrates the inconsistencies that arise when assuming either the chronological or the numerical orderings to detect downgrades or updates. The timeline at the top of the Figure depicts a sequence of releases from the provider and the client packages. For the client package releases, the Figure also shows the used versioning statement and the resolved provider version.

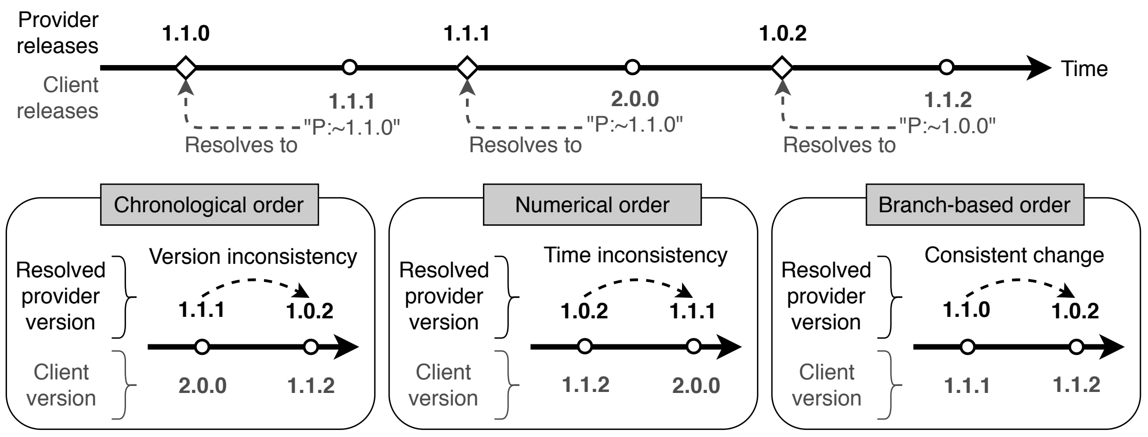


Figure 4.2: Numerical, chronological, and branch-based ordering to evaluate the provider version changes over the client releases.

When the *chronological order* of the client releases is assumed (see Figure 4.2), an incorrect downgrade from version 2.0.0 to 1.1.2 is detected due to a *version inconsistency*. Logically, version 1.1.2 does not succeed version 2.0.0, since these versions belong to different branches. Similarly, when the *numerical order* of the client releases is assumed (see Figure 4.2), an incorrect update from version 1.1.2 to 2.0.0 is detected due to a *time inconsistency*. The provider version resolved at the time that the client version 1.1.2 was released (i.e., provider version 1.0.2) did not exist at the time of client version 2.0.0. Hence, this update is invalid. However, when the *branch-based order* of the client releases is assumed, the changes in the resolved provider version from one client release to another are consistent regarding both version and time.

4.2.2 An algorithm for branch-based ordering

The collected data from npm records only the chronological and numerical orderings of the package releases. Therefore, we conceive an algorithm to derive the branch-based ordering from these two orderings, which works as follows: as the client releases are examined in a chronological order, we check if any of the previously visited releases are in the same branch as the current one (a branch is defined by the major and minor levels, e.g. the 1.0 branch contains versions 1.0.0 and 1.0.1). If so, then the release with the largest version number in the branch of the current release is deemed the predecessor of the current release. Otherwise, the release visited so far with the largest version number is deemed the predecessor of the current release. If the releases in chronological and numerical order, shown in Figure 4.1, are given as input to our algorithm, then the branch-based ordering of the releases (as shown in the same figure) is returned as output.

Algorithm 1 gives the pseudo-code for the algorithm that we conceived to recover the branch-based ordering. The parameters for the procedure BRANCH-RELEASE-ORDERING are R_{T_C}, the list of all releases of a client package C in chronological order, and R_{N_C}, the list of all releases of a client package C in numerical order. This procedure manages sets R and U. The set R stores the pairs of adjacent releases $\langle r_{i-1}, r_i \rangle$ that are identified. In turn, U stores the visited releases from R_{T_C}. The procedure BRANCH(r_i) returns the set of releases that were added to the same branch as that of release r_i. If there are no releases that are added to a given branch, this procedure returns Ø (the empty set). In turn, the procedure UPDATE-BRANCH(r_i, R_b) adds a release r_i to its respective branch (R_b). The procedure LARGEST-VERSION-SMALLER-THAN takes a release r_i and a set L of release versions and returns the largest release version $r_j \in L$ that is smaller than r_i. In case there is no version smaller than r_i in L, the procedure returns a null reference. A given release r_i with a null reference as its predecessor means that r_i has no release preceding it. The procedure LARGEST-VERSION-SMALLER-THAN obtains the precedence between the version number of the releases from the numerical order (R_{N_C}).

Algorithm 1 Sort releases according to a branch-based ordering

```
Input: Releases in chronological ($R_{T_C}$) and numerical ($R_{N_C}$) orders
Output: Releases in branch-based order
  procedure BRANCH-RELEASE-ORDERING($R_{T_C}$, $R_{N_C}$)
      $R \leftarrow \emptyset$
      $U \leftarrow \emptyset$
      for all $r_i \in R_{T_C}$ do
          APPEND($U$, $r_i$)
          $R_b \leftarrow$ BRANCH($r_i$)
          if $R_b \neq$ then
              $r_{i-1} \leftarrow$ LARGEST-VERSION-SMALLER-THAN($r_i$, $R_b$, $R_{N_C}$)
          else
              $r_{i-1} \leftarrow$ LARGEST-VERSION-SMALLER-THAN($r_i$, $U$, $R_{N_C}$)
          APPEND($R$, $\langle r_{i-1}, r_i \rangle$)          ▷ Store $r_{i-1}$ as the predecessor of $r_i$
          UPDATE-BRANCH($r_i$, $R_b$)
      return $R$
```

Algorithm 1: Branch-based release ordering algorithm.

4.3 Data collection

To empirical study on dependency downgrades, we collected data from all packages that were deployed in the npm registry during the period of December 20, 2010 to July 01, 2017. Also, we obtained data from the Version Control System (i.e., the codebase) of a representative sample of packages that perform a downgrade. Four main steps were performed in our data collection procedure: collection of package metadata, analysis of package dependencies, detection of dependency downgrades, and identification of artifacts associated with downgrades. Figure 5.3 depicts an overview of our data collection procedure.

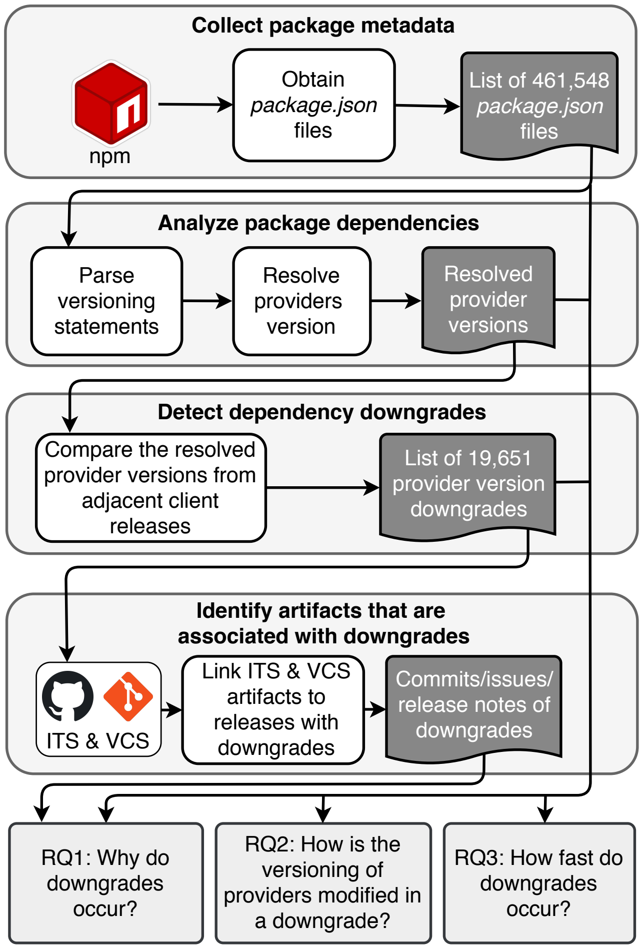

Collect package metadata
npm
Obtain *package.json* files
List of 461,548 *package.json* files
Analyze package dependencies
Parse versioning statements
Resolve providers version
Resolved provider versions
Detect dependency downgrades
Compare the resolved provider versions from adjacent client releases
List of 19,651 provider version downgrades
Identify artifacts that are associated with downgrades
ITS & VCS
Link ITS & VCS artifacts to releases with downgrades
Commits/issues/ release notes of downgrades
RQ1: Why do downgrades occur?
RQ2: How is the versioning of providers modified in a downgrade?
RQ3: How fast do downgrades occur?

Figure 4.3: Overview of our data collection procedure.

4.3.1 Collect package metadata

Obtain *package.json* files: We crawled the registry[6] of npm and obtained the *package.json* metadata file of 461,548 packages. The metadata file of each package lists, among other pieces of information, all the published releases by a client package, the name of the used providers in each release, the associated versioning statements with the providers in each client release, and the timestamp of each release. An example of a *package.json* file can be seen on Chapter 2.

4.3.2 Analyze package dependencies

Parse versioning statements: We parsed the versioning statement of all dependencies in the `package.json` files according to the adopted grammar by npm (see Chapter 2 for a description of such a grammar). Two types of dependencies were considered: (i) production dependencies, which are required for the installation of a package and that are loaded at runtime and (ii) development dependencies, which are not required for the installation of a package and that are loaded only at development time (see Chapter 2 for a description of production and development dependencies).

Resolve providers version: The *package.json* file contains the date on which the releases of a package were published on npm. Given a client package release and the name of a provider that is used in such a release, we initially obtained the list of published versions by the provider before the client release date. Subsequently, we determined the resolved version of each provider according to the versioning statement used by the client in that release (see Chapter 2 for a description of how a provider version is resolved). Dependencies with a versioning statement that did not satisfy any

[6] https://github.com/npm/registry-follower-tutorial

provider version were discarded. The output of this step is the resolved provider version for all client dependencies to providers.

4.3.3 Detect dependency downgrades

Compare the resolved provider versions from adjacent client releases: We sorted the releases of the client packages according to the branch-based ordering algorithm that we discussed in Section 4.2 (see Algorithm 1). Afterwards, we detected downgrades by comparing the resolved provider versions from adjacent client releases. The output of this step is a list of 19,651 downgrades, which were used as input to our RQs. Of all detected downgrades, 48% are from development providers and 52% are from production providers.

4.3.4 Identify artifacts that are associated with downgrades

Link ITS & VCS artifacts to releases with downgrades: From the list of downgrades, it is possible to identify the releases of a client package in which at least one downgrade occurred. In particular, for the packages whose Version Control System (VCS) and Issue Tracker System (ITS) were publicly available, one can also identify which artifacts (e.g., committed files, commit messages) were produced during the releases with downgrades. We identified and examined the artifacts of a statistically-representative sample of client releases with downgrades (more details in Section 4.4.1). The output of this process is thus a list of commits, issues, and release notes that are associated with a sample of the releases that contain at least one downgrade of a provider.

4.4 Results

This section presents the results for each of our RQs. For each RQ, we discuss its motivation, the method that we used to address the RQ, and our findings.

4.4.1 RQ1. Why do downgrades occur?

Motivation: Downgrades indicate that one or more provider versions caused some problem to the client package. Prior studies have only provided limited explanation regarding what these problems are. Therefore, in this RQ, we investigate the rationale behind downgrades.

Approach: We manually examined a statistically representative random sample of client releases in which a provider downgrade occurred. We studied the various artifacts (e.g., release notes, commit messages, and modified files) associated with a client and its release in search for explicit mentions of the rationale for a downgrade. For example, in a commit message that says "*gulp-strip-comment 1.1.1 is broken. force to use an old version*", the rationale for the downgrade is that the package *gulp-strip-comment* at version 1.1.1 caused a failure in the client package.

Figure 4.4 depicts the approach that we used to identify the rationale for downgrades. We initially grouped the list of 19,651 downgrades into the 10,967 client releases in which at least one downgrade occurred. The reason for grouping the downgrades by the client releases is that the distribution of the number of downgrades per client release is skewed: 52% of the downgrades occur in 20% of the client releases with downgrades. Hence, a simple random sampling of the downgrades would be biased towards client releases with many downgrades.

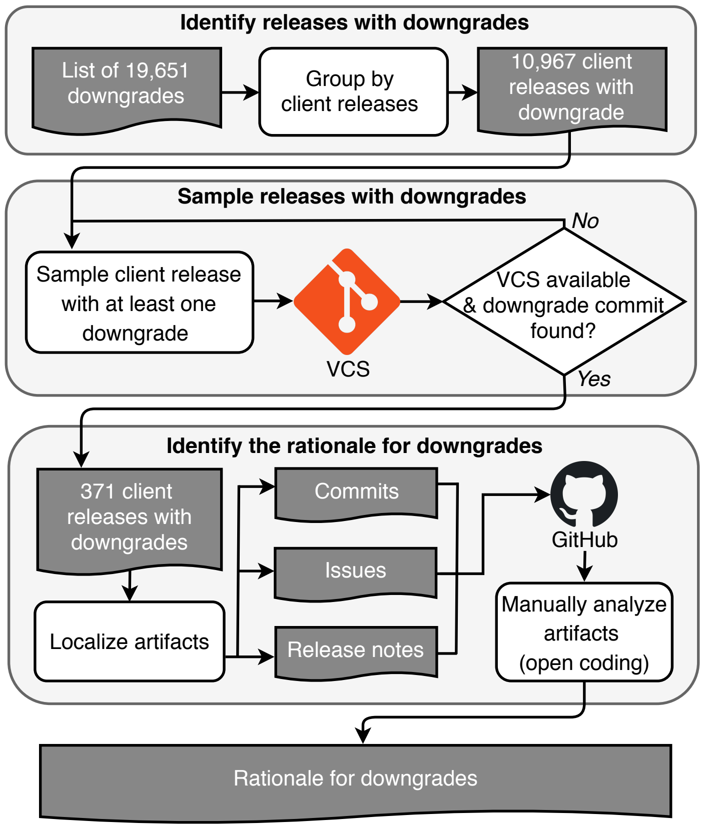

Identify releases with downgrades
List of 19,651 downgrades
Group by client releases
10,967 client releases with downgrade
Sample releases with downgrades
No
Sample client release with at least one downgrade
VCS
VCS available & downgrade commit found?
Yes
Identify the rationale for downgrades
371 client releases with downgrades
Commits
Issues
Release notes
Localize artifacts
GitHub
Manually analyze artifacts (open coding)
Rationale for downgrades

Figure 4.4: Our approach to identify the rationale for downgrades.

After grouping the downgrades by the client releases, we drew a statistically representative sample (95% confidence level and ±5% confidence interval) from these releases (371 cases out of 10,967). For each client package release in our sample of client releases with at least one downgrade, we checked whether the VCS that is used by the package was available and whether the exact commit in which the downgrade was performed could be identified. Whenever an observation in our sample did not meet any of these two requirements, we randomly drew another observation from the population of client releases containing downgrades.

Finally, we manually analyzed the sampled releases in order to identify the rationale for downgrades. The examined artifacts were obtained from the VCS and ITS of each client package. More than 98% of the examined packages used GitHub[7] as their ITS. We performed a thorough examination of the modified artifacts in a commit in which a downgrade was performed. The following artifacts were examined:

- Commit message;
- Artifacts that are modified in the commit, particularly the *package.json* file and the release notes (if available);
- Issues that reference the commit (if available);
- Pull requests that reference the commit (if available).

We performed an open coding (Stol et al., 2016) over the examined artifacts to categorize the rationale for the downgrades. An open coding is a qualitative method to induce general principles that explain a particular instance of a data set. The method generates a set of codes that describes the observed data instances.

[7] http://www.github.com

Observation 4.1) ***Downgrades are performed by client packages either to cope with an issue in a specific provider version or in an attempt to avoid potential future issues.*** This observation led us to separate downgrades in two categories, according to their rationale: *reactive* and *preventive*. The motivation for reactive downgrades is to cope with an issue in a specific provider version that negatively affected the client package. Reactive downgrades are captured by quotes such as "*tar@2.2.1 breaks build*", "*Bluebirds 2.9.x branch has proven to be rather buggy and introduces more issues than it fixes, so lets stick with the stable version*". On the other hand, preventive downgrades are performed to avoid issues from recent provider releases. This latter category is represented by quotes such as "*Locks down package.json dependency versions to avoid build inconsistencies and variation across systems*", or "*We should consider pinning all dependencies to prevent issues like this in the future*".

Observation 4.2) ***There are three issues that motivate a reactive downgrade:*** defect in the provider version, unexpected feature changes in a provider, and incompatibilities. In the following, we describe each of these issues.

Issue 1: ***Defect in the provider version:*** The resolved provider version contains a defect that leads to a failure in the client package. The failure can manifest itself at three different times:

a) Build-time – occurs when the client package fails while it is being installed/built:

"*tar@2.2.1 breaks build.*" [commit message from pull request #103 of package *urllib*],

"*The most recent 0.1.x (0.1.15) broke the build, hence pin it to 0.1.13 for now until it is fixed.*" [message on commit #2b23764 of package *noise-search*],

"*This library currently yield a warning on install*" [discussion on issue #28 of package *ua-parser-json*],

"*Fix version range for devDependencies (...) fixes #28*" [message on commit #2dec1a8 of package *ua-parser-json*]

b) Run-time – occurs when the client package fails while it is running:

"*The update has some breaking changes in how the CircularProgress is rendenred [sic] (...)*" [message on commit #3273dae of package *d2-ui*],

"*lock dependencies so specs run*" [message on commit #1dfad5e of package *wunderbits.db* when downgrading provider *karma*],

"*The way tokens are exposed seems to have changed fundamentally, which broke parsing.*" [message on pull request #832 of package *witheve* when downgrading provider *chevrotain*]

c) Development-time – occurs when the failure manifests itself during the development and in-house testing of the client package:

"*fix versions of things in package.json to original known working versions (trying to get react datum tests working again)*" [message on commit #7397630 of package *bumble-test*],

"*Fixed package.json which for some reason was not allowing webpack and karma validate for my tests.*" [message on commit #5d10a53 of package *karma-styluspreprocessor*]

The defect in the provider might also occur due to degradation of non-functional requirements. The actually resolved provider version is not able to fully adhere to a non-functional requirement of the client package:

"*Downgrade css-loader to 0.14.5 to address superslow HMR builds (...)*" [message on commit #67bec17 of package *brokerjs*],

"*Rollback the Karma dependency version. 'karma' was taking a long time (~30s) to exit the test suites (...)*" [message on commit #6f49232 of package *packery-angular*],

"*Freeze dependencies version to better use cache on Travis (...)*" [message on commit #0d4633d of package *ember-cli-foreigner*]

Issue 2: ***Unexpected feature changes*****:** The current provider version behaves in an unexpected and/or undesired way compared to some prior version (however, the provider's behaviour is not considered defective).

“*Reverting mysql to 2.1.1 (...) Unfortunately mysql has changed the way it handles the charset setting (...) We need to revert this upgrade until the issue is fixed or we have a way to handle it nicely for our users.*” [message on commit #1f17d5b of package *ghost*],

“*Downgraded esdoc to 0.4.3 because they got rid of CLI options*” [message on commit #67bec17 of package *brokerjs*]

Issue 3: *Incompatibilities*: An incompatibility prevents the client package from operating properly. We identified two sources of incompatibilities.

a) Incompatibilities between provider versions – it occurs when the version of two (or more) providers that are used by the client package are not compatible with each other:

“*release 1.3.2 fix fs-extras compability [sic].*” [message on commit #3f0f6c4 of package *yog2-kernel*],

“*Package version modification for compatibility.*” [message on commit #d3f42 of package *mozaik-ext-jira-2* when downgrading package *superagent*],

“*reverted jquery version to 2 for jquery-ui compatability [sic].*” [message on commit #926653a of package *yasgui-yasr*]

b) Incompatibilities with Node version – it occurs when the resolved provider version requires a specific Node version, which in turn is incompatible with the Node version that is used by the client. Node is the run-time engine for JavaScript, which is the language in which npm packages are written:

"*npm versions changed to run the project with node 5.4 (...)*" [message on commit #ee5c524 of package *d3-composite-projections*],

"*Fix npm error on node 0.8.x.*" [message on commit #d5d3078 of package *grunt-wget*],

"*remove caret to allow compatibility with node 0.8*" [message on commit #f0b57e4 of package *nodo*],

"*(...) lock to very specific version that works on node 0.10*" [message on commit #76de1c0 of package *stack-utils-node-internals*]

Observation 4.3) ***A preventive downgrade is performed to avoid potential issues from future releases of the provider.*** This preventive action is often referred to in the examined commit messages as "pinning" or "locking" the provider version. It is done to avoid potential failures that might arise when a provider is updated to a new version. When pinning a provider version, the developer of the client package typically removes the range operator from a version range statement. Such a modification to the versioning statement might lead to a downgrade. For example, if the versioning statement is modified from `"P": ">=2.0.0"` to `"P": "2.0.0"` and the newest version of the provider *P* is 2.0.1, then this provider will be downgraded from 2.0.1 to 2.0.0 when the range operator `">="` is removed. The following excerpts are examples that we observed in the manually examined downgrades:

"*We have now had 2 issues where a "patch" upgrade in a dependency broke Parse Server. (...) We should consider pinning all dependencies to prevent issues like this in the future.*" [discussion on issue #2040 of package *parse-server*],

"*Locks down package.json dependency versions to avoid build inconsistencies and variation across systems.*" [message on pull request #82 of package *lightstep-tracer-javascript*],

"*Use exact versions in package.json: Because some of the new ones caused issue when calling npm install.*" [message on commit #c1e12fe of package *atom-keymap*],

"*Lock broccoli-funnel to prevent rebuild error (...)*" [message on pull request #237 of package *ember-engines*],

"*please pin the moment dependency: By using >= you expose anyone using your package and installing via npm install to different versions of the package being installed.*" [discussion on issue #55 of package *emailjs*],

"*Lock broccoli-funnel to prevent rebuild error (...)*" [message on pull request #237 of package *ember-engines*]

Observation 4.4) ***Preventive downgrades can be triggered by recommendations from automated tools.*** Automated tools that manage dependency versioning can recommend that a client package converts all version ranges to specific versions. These recommendations are deployed through automatically created pull requests that simply remove the range operator from all the versioning statements listed in the client's *package.json* file.

"*Hello! We're all trying to keep our software up to date, yet stable at the same time. This is the first in a series of automatic PRs to help you achieve this goal. It pins all of the dependencies in your package.json, so you have complete control over the exact state of your software.*" [message in pull request #16 of package *noflo-core*]

RQ1: Why do downgrades occur?

- Downgrades occur either as a reaction to a defect in the downgraded provider version or as a prevention to potential issues in future versions of this provider.
- Downgrades usually occur by "pinning" dependencies, i.e., by changing a version range by a specific version statement.

4.4.2 RQ2. How is the versioning of providers modified in a downgrade?

In this RQ, we investigate how the versioning of providers is modified when a provider is downgraded. Our investigation contemplates three different angles, namely: in Section 4.4.2.1, we analyze how the versioning statements are modified in releases containing a downgrade, in Section 4.4.2.2, we study how many providers are downgraded in a release containing a downgrade, and in Section 4.4.2.3 we analyze how the resolved provider version changes when a downgrade occurs.

4.4.2.1 Modification of versioning statements

Motivation: From a client package perspective, using version range statements has the advantage of reducing the overhead of keeping its providers up-to-date. On the other hand, the adoption of version range statements makes the client package susceptible to bugs in provider versions that are implicitly updated. We hypothesize that downgrades are associated with a transition from version range statements to specific versions, specially when a problematic implicit update triggers the downgrade. An evidence of such an association is the occurrence of preventive downgrades and the action of "pinning" dependencies, as we observed in RQ1. In fact, practitioners often advocate against the adoption of version range statements due to the possibility of being caught by surprise by a newly introduced bug in a provider version (Bevacqua, 2015; Draper, 2017; Dodds, 2015). In this RQ, we investigate how downgrades are associated with changes in the versioning statements.

Approach: We calculate the proportion of downgrades that resulted from replacing a version range statement with a specific version and vice-versa. In addition, for the cases in which a version range remains being used after a downgrade, we investigate how the operators and numerical part of the versioning statement are modified.

Observation 4.5) ***Almost half (49%) of the downgrades occur due to a replacement of a version range statement with a specific version.*** Figure 4.5 shows the proportion of downgrades per type of versioning statement change. The most common type of versioning statement change in downgrades (49%) is *from range to specific* (of which 49% are from production providers). In this subgroup, 68% of the cases were performed simply by removing the range operator and keeping the numerical part. As we observed in RQ1, this is how preventive downgrades are typically performed: instead of

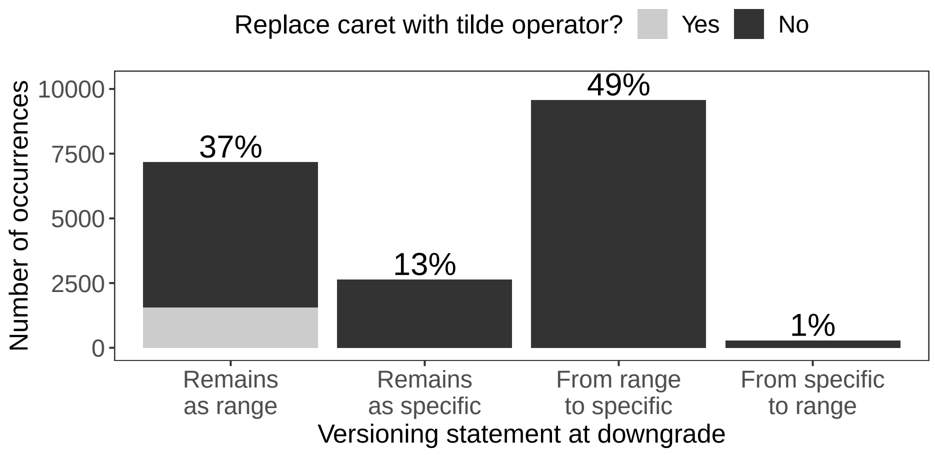


Figure 4.5: Proportion of downgrades per type of versioning statement change.

carefully choosing a provider version, developers simply remove the version range operator from versioning statements. Also, when the versioning statement remains as specific, the proportion of downgrades that are from production providers is 61%. In turn, the same proportion is 48% when the versioning statement changes from specific to range.

As also shown in Figure 4.5, the versioning statement *remains as a range* in 37% of the downgrades (of which 51% are from production providers). In this subgroup, 21.8% of the cases involved replacing a caret (∧) operator with a tilde (∼) operator. The tilde operator resolves towards a patch update of the provider, while the caret operator resolves towards a minor update. In 58.1% of the caret-to-tilde replacements, the numerical part of the version range did not change. As a consequence, the range of provider versions that are accepted by the client is narrowed down. There are no cases for which the change from caret to tilde would be effectless (i.e., downgrades in which the versioning statement change from `"∧0.x.y"` to `"∼0.x.y"`).

4.4.2.2 Number of providers that are downgraded in a release

Motivation: When one or more providers cause an issue, client package developers might perform a reactive downgrade. However, detecting the specific troublesome provider might not be trivial and developers might end up downgrading unrelated providers. In addition, "pinning" a large number of providers might result in downgrading many providers at once. Hence, in this research question we investigate how localized downgrades are.

Approach: We categorize all client releases with downgrades as having one, two, three, four, five, and more than five providers downgraded. Afterwards, we count the number of client releases with downgrades that fit these categories. For each client release with a downgrade in one of these categories, we verified the proportion of providers (from the total number of used providers) that were downgraded.

Observation 4.6) ***In 75.5% of the client releases containing a downgrade, only a single provider is downgraded in these releases.*** This observation is depicted in Figure 4.6. In 58.3% of the releases in which a single provider was downgraded, more than 10 providers were being used by the client package (black portion of left-most stacked bar). On the other hand, in only 2.5% of all releases with downgrades all providers were downgraded at once (sum of lightest gray portion over all stacked bars). These results thus indicate that downgrades are often localized.

4.4.2.3 Introduced technical lag

Motivation: When a client package performs a downgrade, there is an increase in the technical lag regarding the resolved provider version (Gonzalez-Barahona et al., 2017;

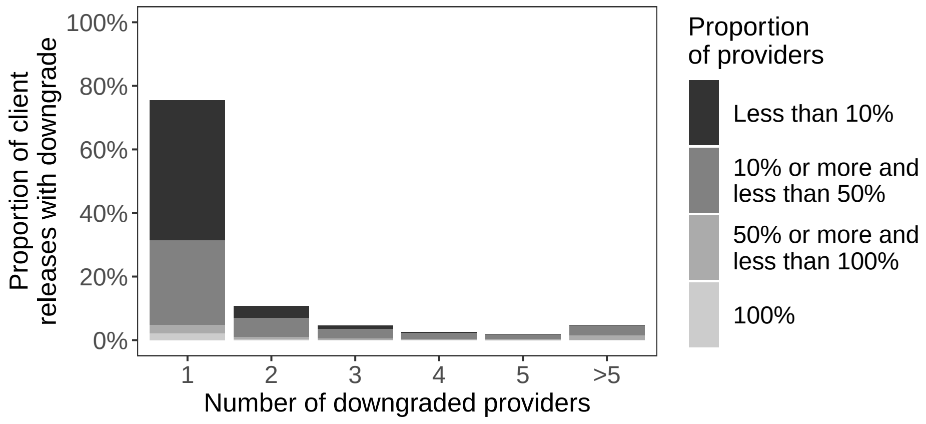


Figure 4.6: Number of downgraded providers in the same client release.

Decan et al., 2018). As such, downgrades naturally prevent client packages from leveraging the benefits brought by newer releases, including bug fixes, vulnerabilities fixes, and new features (Bavota et al., 2015; Cox et al., 2015; Derr et al., 2017). Thus, it is generally advised that client packages keep their providers up-to-date. The technical lag introduced in a downgrade can not only affect the client package itself, but also affect transitive dependencies. For this reason, it is important to evaluate the impact of downgrades in other packages in the ecosystem. In this RQ, we measure the impact of downgrades on the technical lag and the extent to which the introduced technical lag can impact client packages that use a release with downgrade. Finally, Zerouali et al. (2019b) show that there is a difference between the technical lag of development and production providers. Therefore, we study whether the technical lag introduced in a downgrade differs between these two types of providers.

Approach: We calculate the proportion of provider versions that were reduced by a major, minor, and patch level in a downgrade. In addition, we compare the increased technical lag when a downgrade occurs with the decreased technical lag when the prior update occurred. The increase (or decrease) in technical lag is measured by the number of

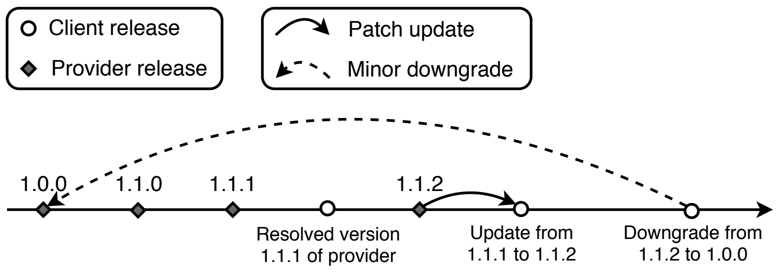

Client release
Provider release
Patch update
Minor downgrade
1.0.0
1.1.0
1.1.1
1.1.2
Resolved version 1.1.1 of provider
Update from 1.1.1 to 1.1.2
Downgrade from 1.1.2 to 1.0.0

Figure 4.7: Number of back skipped provider versions in a minor downgrade following a patch update.

already published provider versions that are back (or forward) skipped in a downgrade (or update), according to the numerical ordering. Figure 4.7 depicts this calculation. The figure shows an update followed by a downgrade. A patch update changes the resolved provider version from 1.1.1 to 1.1.2, decreasing the technical lag by one patch release. On the next client release, the fourth version (1.1.2) of the provider is downgraded towards the first version (1.0.0). In this example, the downgrade back skipped three versions (1.1.2, 1.1.1, and 1.1.0). Hence, we say that the technical lag was increased by three versions (or two patch and one minor release).

We verify if the distribution of the number of backskipped major, minor, and patch releases is different between downgrades of development and production providers. To compare the distributions, we test the null hypothesis that both distributions do not differ from each other using the Wilcoxon Rank Sum test ($\alpha = 0.05$) (Bauer, 1972) and assess the magnitude of the difference with the Cliffs Delta (d) estimator of effect size (Cliff, 1996). To classify the effect size, we use the following thresholds (Romano et al., 2006): *negligible* for $|d| \leq 0.147$, *small* for $0.147 < |d| \leq 0.33$, *medium* for $0.33 < |d| \leq 0.474$, and *large* otherwise.

Observation 4.7) ***13% of the downgrades induce an unnecessary increase in the technical lag.*** Table 4.1 shows the proportion of patch, minor, and major downgrades that

Table 4.1: Proportion of major, minor, and patch downgrades that follow a major, minor, or patch update.

		Update that precedes the downgrade		
		Patch	Minor	Major
Downgraded version level	Patch (27%)	95%	3%	2%
	Minor (48%)	18%	80%	2%
	Major (25%)	9%	9%	82%

follow a patch, minor, and major update. Downgraded version levels that are larger than the updated version level are shown in grey filled cells. Almost one fifth of the minor downgrades (18%) follow a patch update and a total of 18% of the major downgrades follow a patch or a minor update. For such cases, the downgrade not only nullifies the benefits of the prior update, but also increases the technical lag.

In 48% of the downgrades, the provider package version is reduced by a minor level. Patch and major downgrades represent, respectively, 27% and 25% of the downgrades. Interestingly, these proportions do not correspond to the proportion of patch, minor, and major releases of downgraded packages which are, respectively, 72%, 23%, and 5%. In addition, almost one fifth (19.4%) of the releases containing a downgrade have at least one client package, representing cases in which the technical lag can affect transitive dependencies.

Observation 4.8) ***Major downgrades back skip a median of 1 major, 1 minor, and 3 patch releases.*** Figure 4.8 shows the increase in technical lag for each modified version level in a downgrade. In 65% of the major downgrades, at least one minor release is also back skipped, while in 91% a patch release is also back skipped (59% back skip both minor and patch releases). Also, we verified that 85% of the minor downgrades back

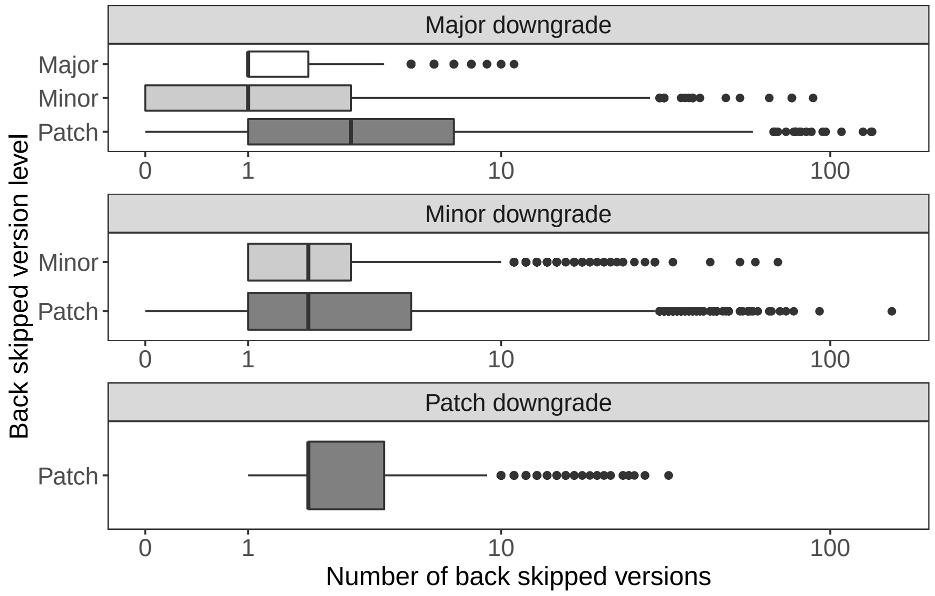


Figure 4.8: Number of back skipped provider versions in a major, minor, and patch downgrade.

skip at least one patch release. Comparing downgrades of development and production providers, the difference between the distribution of the number of backskipped major and patch releases is not statistically significant (p-value > 0.05). While the difference for the number of back skipped minors is statistically significant, the effect size is negligible ($|d| = 0.114$).

Major downgrades represent one quarter of all downgrades and typically follow a pattern. 82% of the major downgrades are preceded by a major update. Also, 75% of the updates preceding a major downgrade are explicit, indicating that the versioning statement used at the update time generally does not satisfy the new major releases of the provider and that major updates tend to be a well-thought-out decision. Finally, 70% of the major downgrades are rollbacks (i.e., the target of the downgrade is the version that was originally used by the client package). These observations suggest that major downgrades are likely the result of a failed attempt to update to a major version.

In 88% of the major updates that precede a major downgrade, the update was explicit. This observation shows that the majority of the major updates that precede a major downgrade are deliberated, suggesting that many issues that arise after a major update of a provider manifest themselves in-field (i.e., after deployment) but not in-house (i.e., at the development environment).

RQ2: How is the versioning of providers modified in a downgrade?

- Downgrades are associated with a shift into more conservative versioning of providers: 49% of the downgrades replace a version range by a specific version of the provider.
- Downgrades are generally localized: In 75% of the client releases containing a downgrade, a single provider is downgraded.
- 13% of the downgrades induce an unnecessary increase in the technical lag.

4.4.3 RQ3. How fast are downgrades performed?

Motivation: A downgrade indicates that one or more providers caused some issue to the client package. In particular, when the provider is implicitly updated (i.e., because the new provider version satisfies the specified version range), these issues can manifest themselves in a sudden manner, making it challenging to rapidly identify the provider that is associated with the issue. In fact, prior research has shown how unexpected defects impact the quality of a software product (Shihab et al., 2011). Measuring the time between the update of a provider version and the consequent downgrade can thus help to understand how fast client packages are able to react to the issues behind downgrades. Furthermore, client packages can react to issues that arise from development and production providers with a different degree of urgency. For example, issues from development providers should not impact the deployed client package and the contingency of such issues can be delayed without affecting the client package's users. In this RQ we also differentiate between the time to downgrade development and production providers.

Approach: To determine how fast a downgrade is performed, we calculate the ratio shown in Equation 4.1. The ratio is a means to compare the taken time to perform a given downgrade with the typical time between two releases of a client package. Values larger than 1 indicate that a downgrade D takes more time to occur than a typical release of the client package C. A typical time between releases of a client package C is calculated by $\mathit{timeRel}(C)$ in Equation 4.1. We verify whether there is statistical difference between the $\mathit{speedRatio}(C,D)$ of development and production providers. To do so, we used the Wilcoxon Rank Sum test ($\alpha = 0.05$) and the Cliff's Delta estimator of effect size. To classify the effect size, we use the following thresholds (Romano et al., 2006): *negligible* for $|d| \leq 0.147$, *small* for $0.147 < |d| \leq 0.33$, *medium* for $0.33 < |d| \leq 0.474$, and *large* otherwise.

$$\mathit{speedRatio}(C,D) = \frac{\mathit{time}(D)}{\mathit{rel}(D) \times \mathit{timeRel}(C)} \tag{4.1}$$

where:

- $\mathit{time}(D)$ is the elapsed time (in days) between the update and the eventual downgrade D.

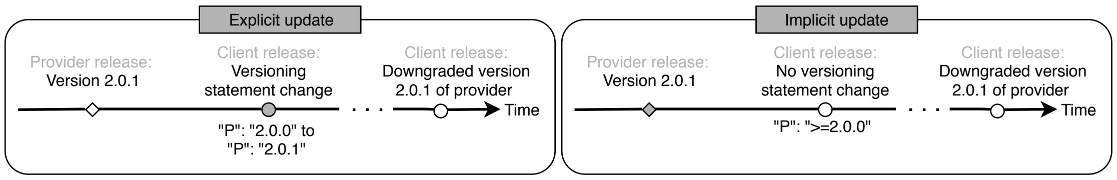

Explicit update
Provider release: Version 2.0.1
Client release: Versioning statement change
Client release: Downgraded version 2.0.1 of provider
Time
"P": "2.0.0" to "P": "2.0.1"
Implicit update
Provider release: Version 2.0.1
Client release: No versioning statement change
Client release: Downgraded version 2.0.1 of provider
Time
"P": ">=2.0.0"

Figure 4.9: Downgrades preceded by explicit and implicit updates.

- $rel(D)$ is the number of spanned client releases between the update and the eventual downgrade D (inclusive).
- $timeRel(C)$ is the median elapsed time between the last half releases of a client package C (in days per release).

We also investigate whether downgrades of implicit updates take longer than downgrades of explicit updates. Given a downgrade, we determine the timestamp of the preceding update based on how this update occurred. When an update is explicit, i.e., it occurs because the versioning statement was modified, the timestamp is the date at which the client package publishes a release with the updated provider (depicted as a shaded dot in the timeline on the left-hand side of Figure 4.9). On the other hand, when an update is implicit, i.e., it occurs because the actual version range satisfies the new version of the provider, the update timestamp is the date at which the provider released the version that was eventually downgraded (depicted as a shaded diamond in the timeline on the right-hand side of Figure 4.9).

We compare the distribution of the elapsed time between an explicit update and its eventual downgrade with the distribution of the elapsed time between an implicit update and the eventual downgrade. This comparison is controlled by the downgraded version level (i.e., whether it is a patch, minor, or major downgrade). We compared the

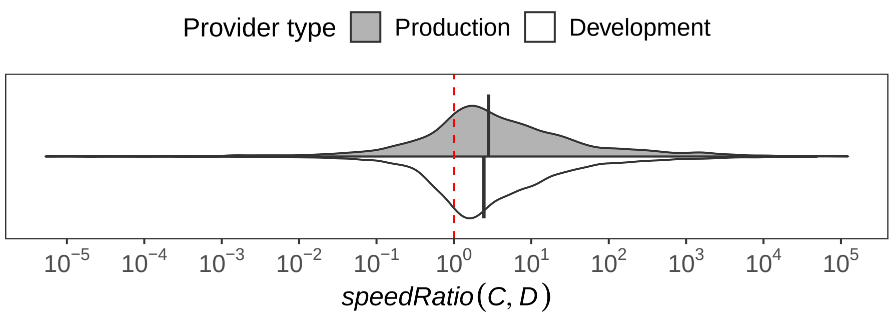


Figure 4.10: Distribution of the *speedRatio*(C, D). The red dotted line indicates a ratio value of 1.

distributions using the Wilcoxon Rank Sum test ($\alpha = 0.05$) and the Cliff's Delta estimator of effect size (see Section 4.4.2.3 for a classification of the effect size).

Observation 4.9) ***50% of the downgrades are performed 2.6 times slower than k typical releases, where k is the number of releases taken for the downgrade to occur.*** Figure 4.10 shows the distribution of the *speedRatio*(C, D) (Equation 1) for development and production providers. The difference between the two distributions is statistical significant ($p < 0.05$) with a negligible effect size ($|d| = 0.026$). The first quartile of both distributions is greater than 1, indicating that more than 75% of the client releases published during the update and following downgrade of a provider are slower than the typical releases of the client package. In addition, Table 4.2 shows the median, mean, first, and third quartile for the variables used in Equation 4.1. The median time for a downgrade to occur is 34.8 days and 50% of the downgrades occur in one or two releases after the update of the provider.

Observation 4.10) ***Downgrades of an implicit update generally take longer than downgrades of an explicit update.*** The observed difference can be explained by the fact that implicit updates are not controlled by client package developers. Hence, an issue that arises after the provider is updated can appear unexpectedly. Thus, developers might

Table 4.2: Summary of the variables calculated in Equation 4.1.

	1st quart.	Median	Mean	3rd quart.
$speedRatio(C,D)$	1.02	2.63	191.59	10.78
$time(D)$	10.73	34.84	79.76	95.86
$rel(D)$	1.00	2.00	5.05	4.00
$timeRel(C)$	1.24	5.55	19.14	18.14

need additional time to identify the provider(s) that is(are) associated with the issue that concerns the downgrade. On the other hand, when performing an explicit update, developers are aware of which providers were modified, making it easier to identify a provider that eventually needs to be downgraded.

Considering explicit updates that precede a downgrade, the median time to downgrade a provider is 35 days for major downgrades, 37 days for minor downgrades, and 46 days for patch downgrades. Considering implicit updates, the median time is, respectively, 36, 48, and 35 days for major, minor, and patch downgrades. Figure 4.11 depicts the distributions. All pairwise differences between the downgraded version levels (grouped by implicit and explicit updates) are statistically significant, but have negligible effect size. In turn, comparing the update types (implicit vs. explicit) for major, minor, and patch downgrades, we obtain, respectively, a statistically significant difference with large effect size ($|d| = 0.474$), medium effect size ($|d| = 0.454$), and large effect size ($|d| = 0.477$).

Observation 4.11) ***Urgent downgrades occur more often after an explicitly update than after an implicit update.*** Urgent downgrades refer to downgrades that occurred in less than one day after the update of the provider. Figure 4.11 depicts this observation. We found that 5.6% of all downgrades are performed in an urgent manner. Also, 37.1% of the downgrades of explicit updates are urgent downgrades. In contrast,

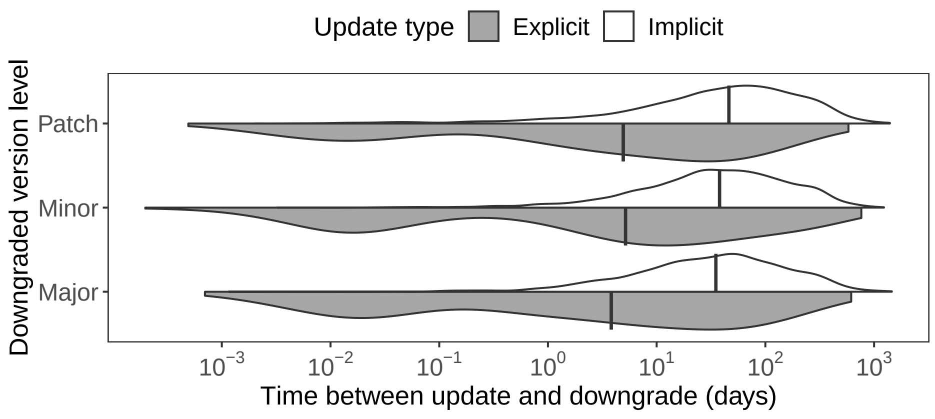


Figure 4.11: Distribution of the elapsed time between the update and the eventual downgrade of a provider package.

only 3.8% of the downgrades of implicit updates are urgent updates. This observation corroborates our conjecture that, because explicit updates are controlled by the client packages, developers are more likely to react fast to the issues that were brought by these updates. In addition, 67% of the urgent downgrades are from production providers.

RQ3: How fast are downgrades performed?

- At least 75% of the releases with a downgrade are slower than the typical release
- 37% of the explicit updates are downgraded within 24 hours
- Downgrades of implicit updates are slower than downgrades of explicit updates

4.5 Discussion

In this section, we discuss the implications that derive from the results of our RQs. The general learned lesson from our empirical findings is that *package developers should improve the management of their dependencies.* In particular, package developers should keep track of their dependencies over time and be cautious with provider updates. Ultimately, these practices should optimize the debug of troublesome updates (consequently reducing the time to fix issues that affect the client packages) and reduce the technical lag that is introduced when a downgrade occurs. In the following, we present specific implications to help practitioners manage dependencies.

Implication 4.1) ***Package developers can use automated tools to support early discovery of provider issues and thus decrease the time taken to downgrade.*** The issue that motivates a downgrade can take some time to manifest itself. In particular, downgrades associated with implicit updates take longer to occur (c.f., Observation 10), thus delaying the provision of the fix to the packages' users. Several tool-assisted approaches can be employed to support the *early* detection of troublesome provider updates. A simple approach can be employed using three different tools: (i) latest-version[8] checks what the latest version of a provider package is, (ii) next-update[9] runs the client's test suites, and (iii) npm-check-updates[10] automatically updates the versioning statements in the *package.json* file A step further in the degree of automation involves the usage of bots to manage dependency updates, such as Greenkeeper[11] and Renovate[12]. These bots interact with client package developers through the package's ITS (e.g., GitHub). The following workflow is generally implemented: (i) the bot identifies an opportunity for an implicit update of a provider, (ii) the implicit update is performed in an isolated branch, (iii) the bot runs a suite of automated tests and attempts

[8] https://www.npmjs.com/package/latest-version
[9] https://www.npmjs.com/package/next-update
[10] https://www.npmjs.com/package/npm-check-updates
[11] https://greenkeeper.io
[12] https://renovatebot.com

to rebuild the package, (iv) in case the test suite or the build fails, the bot opens an issue report with recommended actions.

Implication 4.2) ***Client packages can log their dependency tree to debug troublesome providers.*** While the approach described in Lesson learned 1 ensures that tests pass and the build does not break, it is still possible that a provider package might lead to a problem. For instance, incompatibilities or a performance regression might not be captured by the packages test suite. In this scenario, package developers might need to debug the troublesome update. A lightweight approach would consist of simply keeping a log file containing the state of the client package's dependency tree (i.e., all the provider versions that are loaded at a given time). With such log files, developers can trace back the state of the dependency tree at a given time and determine the exact updates that potentially led to the problem. Logging updates can be achieved by setting an automated background routine that uses simple commands provided by npm and the VCS. Such a routine can be implemented by: 1) using the npm-update command in an isolated environment (e.g., in a new branch or within any folder created to this end), such that this isolated environment contains the client package with its providers updated to the latest version that satisfies the versioning statements, 2) using the npm-ls command to produce a report of the dependency tree with the loaded provider versions, 3) committing the dependency tree state on a daily basis to the VCS, and 4) using some VCS's tool (e.g., blame[13] from Git) to identify the update of a given provider. This workflow can be automated by setting a periodic routine that performs the four aforementioned steps.

Implication 4.3) ***Client packages using a flexible dependency versioning strategy (i.e., extensive use of range statements) should emphasize testing the functionalities that***

[13] https://git-scm.com/book/it/v2/Git-Tools-Debugging-with-Git

involve provider packages. Reactive downgrades are caused by issues coming from the provider packages. Therefore, testing functionalities that rely on the providers should be intensified. In addition, testing corner cases for the provided functionalities by the providers can safeguard client packages, especially when those functionalities are not fully tested by the provider itself. Moreover, due to incompatibilities, client packages should, if possible, test scenarios where multiple providers are used together.

Implication 4.4) ***Client packages should be mindful of the latest working provider version when pinning a dependency.*** Almost half (49%) of the downgrades are performed by pinning the provider version. Pinning is typically (68%) performed by removing the range operator and keeping the numerical part of the range statement (c.f., Observation 5). However, this downgrade pattern can lead to the adoption of a version of the provider that is older than the latest working provider version (thus increasing technical lag). Therefore, client packages should consider the latest working version of the provider instead of simply removing the version range operator from the versioning statement. In fact, the own package manager could detect the removal of a version range operator and warn developers of its side effects (e.g., unnecessary technical lag being introduced).

4.6 Related work

In this section, we describe related work concerning dependency downgrades. Two prior studies mention the phenomenon of downgrades in the npm ecosystem. Decan et al. (2018) analyzed the impact of security vulnerabilities in npm dependencies. The authors highlight that vulnerable providers impact the quality of their client packages.

The authors also claim that vulnerability issues can be solved by "rolling back to an earlier version" of a provider. When analyzing a representative sample of the downgrade cases (Section 4.4.1), we did not identify any explicit mentions to downgrades being performed because of vulnerabilities in a provider. However, we did identify that one of the rationales for downgrades is the presence of defects in the provider. In this sense, it is possible that these defects encompass vulnerabilities issues.

Mirhosseini and Parnin (2017) studied the effectiveness of automated tools (e.g., bots) to manage dependencies in npm. The authors show that providers are updated 1.6 times more often and 1.3 times faster when client packages use such bots, compared to clients that do not. By means of a survey, the authors demonstrate that the three most common developers concerns regarding the update of providers are backward-incompatible changes, understanding of the implications of changes, and migration effort. Furthermore, 24% of the builds fail when some provider version is changed. The authors also found that several provider updates performed by bots were downgraded in 2 or 3 days after being merged. In this thesis, we found that defects in the provider, in particular the ones that affect the client package build, motivate downgrades. In addition, we identified that bot recommendation is one of the reasons behind preventive downgrades of a provider.

A few other papers approach the subject of downgrades in software ecosystems other than npm, such as the Apache[14] and the Android[15] ecosystems. Mileva et al. (2009) performed an empirical study of over 250 Apache projects with the goal of understanding how the popularity of a package relates to its quality. They propose that the number

[14]https://www.apache.org
[15]https://developer.android.com

of downgrades of a given provider is an indicator of the (lack of) quality of that package. Our findings corroborate this proposition.

Salza et al. (2018) analyzed the categories of provider packages that were downgraded in mobile apps. They found that Graphical User Interfaces (GUI) and Utilities are the categories of providers with the highest number of downgrades. The authors show evidence that client packages want to follow look and feel tendencies, which explains the high number of updates of GUI-related providers. Also, utility packages support the development of applications in the ecosystem and are highly popular. Despite the high number of packages that depend on those providers in mobile apps, the authors did not explain *why* downgrades of these providers are often performed.

4.7 Threats to validity

Internal validity: When identifying the reasons for a downgrade, we searched for the specific commit in which the provider versioning was changed in the *package.json* file. However, it is possible that the downgrade reason was revealed in some prior commit. We likely missed those cases in our analysis. Furthermore, in our manual analysis, we did not inspect every file in the commit but merely searched for an explicit mention for a downgrade in the examined artifacts (see method in Section 4.4.1). Also, we acknowledge that different classifications of the downgrades (e.g., based on prior theories about maintenance categories or derived from interviews with developers) would likely yield a complementary view to our results.

The proportion of published major, minor, and patch releases are different across npm packages. Such non-uniform distribution of releases types has an impact on the interpretation of our results. We mitigate this threat by controlling for release type (i.e., major, minor, and patch) when appropriate (Sections 4.4.2 and 4.4.3).

External validity: Because we collected data exclusively from npm, our findings might not be generalizable to other ecosystems. Although npm is representative in size, in fact, each software ecosystem has its own intrinsic characteristics. The goal of this study is not to build theories around downgrades that would apply to all software ecosystems. Rather, our study is only a first step towards a deeper understanding of why and how packages are downgraded. Therefore, we acknowledge that additional studies are required in order to further generalize our results. Nonetheless, to the best of our knowledge, this is the first study to thoroughly investigate the phenomenon of downgrades. In addition, our approach can be replicated in other ecosystems. Structures similar to versioning statements (i.e., that allows one to set a specific version or a range of versions to a given provider) and version numbering schemes can be found in several other software ecosystems, such as Bundler (for Ruby), Cabal (for Haskell), pip (for Python) and Maven (for Java). Downgrades of provider versions can also be found in all those ecosystems platforms.

Construct validity: We identified the downgrades in npm based on a heuristic for ordering the client releases, which we call branch-based order (see Section 4.2). However, although this heuristic is a best-effort to capture the logical ordering of the releases, the actual ordering adopted by a package can be arbitrary. Therefore, it is not guaranteed that the use of such heuristic captures all downgrades performed in npm. Nevertheless, we consider that, with respect to the identification of the downgrades, the branch-based ordering represents the actual order of releases more accurately than either the numerical or chronological orderings. The reasons for this consideration are explained in Section 4.2. Furthermore, our manual analysis served as a sanity-check for the reliability of our approach to detect downgrades. Finally, in Section 4.4.3, we calculate the typical inter-release time of a package as the median time between the last half releases. Although considering the last half releases is an arbitrary decision, this is a reasonable form of representing the current inter-release time of packages that have different release schedules.

Swanson (1976) proposes four dimensions of maintenance activities, namely corrective, adaptive, perfective, and preventive. The classification of downgrades into reactive and preventive (RQ1) is related with the dimensions of maintenance proposed by Swanson (1976). Conceptually, reactive downgrades can be understood as a combined form of corrective, adaptive, and perfective maintenance, while preventive downgrades can be understood as a preventive maintenance activity. However, for some of the manually investigated cases, it was impossible to determine, given the available evidence, whether a reactive downgrade was corrective, adaptive, or perfective. Hence, driven by the constraints of the data that we investigated, we simply classify downgrades into reactive and preventive.

4.8 Conclusions

The benefits of having up-to-date provider versions is extensively studied in the literature (Bavota et al., 2013; Ruiz et al., 2016; Kula et al., 2017b). Prior studies also point to the reasons why developers might prefer not to update provider packages (Kula et al., 2015; Bavota et al., 2015; Derr et al., 2017). On the other hand, only a few papers examine aspects related to downgrades in software ecosystems (Mileva et al., 2009; Salza et al., 2018; Decan et al., 2018). Using historical data from package releases, we empirically investigate downgrades in npm. In particular, we study why provider packages are downgraded, how they are downgraded, and how fast the downgrade occurs. Our results show that downgrades are a facet of the management of dependencies in software ecosystems, being used as a workaround to deal with issues coming from provider packages. In Section 4.5 we discuss a set of procedures that practitioners can implement to better cope with the need for downgrading a provider. We make the following observations:

- *Downgrades are performed because of issues that arise from the provider, but are also performed for preventive purpose.* We identified two types of downgrades as reactive and preventive (Observation 1). Three issues motivate reactive downgrades, namely defects in the provider version (during build-time, run-time, or development-time), sudden feature changes in the provider, and incompatibilities (between provider versions, or with Node version) (Observation 2). On the other hand, preventive downgrades are originated from the preventive action often called by client developers as "pinning" the provider version (Observation 3). Also, preventive downgrades can be triggered by bot recommendations (Observation 4).

- *Downgrades are associated with a change to a more conservative versioning of providers.* We observed that 49% of the downgrades change the versioning statement from range-to-specific. In addition, when the versioning statement remains as range after a downgrade, the range of acceptable versions is often narrowed down (Observation 5). In 75.5% of the client releases with downgrade, a single provider is downgraded (Observation 6). 13% of the downgrades induce an unnecessary increase in the technical lag (Observation 7). Nonetheless, downgrades of major versions also occur and they normally introduce a larger technical lag on the client package compared to minor and patch downgrades (Observation 8). An explanation is that client packages often delay the integration of major releases of a provider due to its inherent increased difficulty compared to minors and patches.

- *The speed with which a downgrade occurs is associated with how the provider was formerly updated.* Client releases published between the update and the following downgrade of a provider take 2.6 times longer than the typical time-between-releases of the same client package (Observation 9). More than half of the downgrades that follow an explicit update are performed almost 10 times faster than half of the downgrades following an implicit update (Observation 10). Also, we observed the occurrence of urgent downgrades, i.e., those that occur up to 24 hours after the prior downgrade. There are almost 9 times more urgent downgrades following an explicit update (36%) than urgent downgrades following an implicit update (3.8%) (Observation 11).

Our observations contribute to advancing the research concerning dependency management in software ecosystems. In particular, our observations complement prior studies that relate downgrades to issues in the provider packages, but that did not describe the nature of those issues. Based on the identified causes for downgrades and on the understanding of how downgrades are commonly performed, we derived a set of learnt lessons to help client packages mitigate the side-effects of downgrades. Lastly, we outline future research opportunities.

CHAPTER 5

An Empirical Study of Same-day Releases of Popular Packages

In a software ecosystem, where a large number of dependencies between client and provider packages exist, a problematic release of a popular provider package can affect a large number of clients. When detected early, such problematic releases can lead to the development of a same-day release (a time-constrained corrective release published in the same day as the previous releases). In this study, we consider same-day releases that are published in the npm *ecosystem by popular packages (i.e., packages with a significantly large number of client packages than others). We found that 20% of the patch releases of popular packages are same-day. Almost one-quarter of such same-day releases modify more lines of code than the regular releases of the same package. In general, although the rate at which same-day releases are explicitly updated by client package is comparable to the update rate of regular releases, explicit updates to same-day releases are significantly faster than updates to regular releases. The results presented in this study can help development teams, in particular release managers, to make informed decisions regarding their same-day releases.*

An earlier version of this chapter is under major review in the Empirical Software Engineering Journal (EMSE).

In this chapter, we describe our study about same-day releases in npm. Section 5.1 introduces and motivates our study. Section 5.2 presents background and related work regarding same-day releases. Section 5.3 explains our data collection methodology. Section 5.4 presents our preliminary study of changes introduced in same-day releases by analyzing same-day release notes. Section 5.5 presents the results of our research questions. Section 5.6 discusses the implications of our findings and draws a set of actionable insights for development teams. Section 5.7 discusses the threats to the validity of our study. Finally, Section 5.8 states the conclusion remarks.

5.1 Introduction

Occasionally, multiple releases of a software project need to be published on the same day. Prior studies associate those same-day releases with a time constrained response to an issue that was caused by the prior release. For instance, Adams et al. (2015) present that release engineers in Firefox publish "zero-day fixes" in response to failed releases. Similarly, Kerzazi and Adams (2016) study the releases of a web-based commercial system that fail "a few minutes" after being published, consequently requiring the development team to timely react to the issue. In a software ecosystem, code reuse is enabled by means of dependencies between packages. If a package c specifies a dependency $D_{c,p}$ on another package p, then c can reuse the code of p. In this study, we refer to c as a client package and p as a provider package, respectively. Same-day releases are especially relevant in software ecosystem platforms, where many client packages may depend on specific releases of a provider package. For example, mocha is a popular package in the npm ecosystem, with more than 120K client packages and 4M weekly installations. Version 2.5.0 of mocha was released with a defect caused by a specific dependency,[1] preventing the installation of version 2.5.0 of the mocha package by any of its clients. On the same day, version 2.5.1 was released to fix the dependency issue. Another example is the defect caused by a typo introduced in version 2.4.0 of the also popular debug package (Mezzetti et al., 2018).[2] The defect affected a large number of client packages and a same-day release was published to address the problem. Such examples denote the importance of same-day releases for maintaining a stable ecosystem.

The time available to revise, test, build, and document same-day releases is restricted compared to releases published within a regular schedule. When the defect on version 2.5.0 of the mocha package was identified and fixed, the build time was a concern, as expressed by a client package developer in an issue report: "*How long for build? Can one check progress somewhere?*". A time related concern could also be observed when the typo in the debug package was reported: "*Please tone it down, everybody. Mistakes happen. Its been about 30–40 minutes, and its being worked on right now.*" As a consequence of the inherent time constraint of same-day releases, developers might need to prioritize specific activities (such as testing or building specific parts of the system). For example, Castelluccio et al. (2019) study the practice of patch uplifting by the Firefox development team. The authors observe that, occasionally, a well-established release schedule needs to be disturbed by an urgent release that contains critical fixes. Such urgent releases are "uplifted" from a development to a stable channel without being thoroughly checked. Therefore, the appropriate management of same-day releases is an important aspect to be considered by developers.

[1] https://github.com/mochajs/mocha/issues/2276
[2] https://github.com/visionmedia/debug/issues/347

Although prior studies examined related phenomena with same-day releases, the software ecosystem context was never considered in such studies. Same-day releases are particularly relevant in software ecosystems because there is an ever-changing network of dependencies among packages. In particular, many client packages start adopting a provider's release automatically, due to the usage of version range statements (i.e., the default npm versioning mechanism, which allows a client package to set a range of provider versions that are automatically updated). By automatically adopting new provider versions, client packages can be caught off-guard when failures appear. Nevertheless, basic information about same-day releases is still unknown, such as the frequency with which same-day releases occur, how same-day releases are documented, what the typical changes in same-day releases are, and how they impact client packages. In this chapter, we empirically investigate the occurrence and adoption of same-day releases in the npm ecosystem. We focus on the following research questions:

Preliminary study. To understand the characteristics of the introduced changes in same-day releases, we perform a qualitative analysis over a random sample of same-day release notes. Our analysis resulted in a fine-grained set of categories that describe introduced changes in same-day releases, ultimately revealing their characteristics and scope. We observed that the contents of the same-day release notes point to relevant changes, mainly bug fixes. The introduced changes in same-day releases can be related to the package's own business logic or to the integration with other provider packages. We also observed that introduced changes address critical issues, such as crashes, UI failures, and performance degradation.

RQ1. How often do same-day releases occur? Same-day releases represent 26% of all studied releases and are published at least once by 93% of the studied packages in npm. Same-day releases are more common for a specific set of the studied packages: 66% of all studied same-day releases are published by 25% of these packages. Also, more than one-third (39%) of the studied same-day releases are followed by another same-day release.

RQ2. What are the performed code changes in same-day releases? We observed that 32% of the same-day releases modify more lines of code compared with its prior release. We also observed that, even with the inherent time constraint, provider packages are updated in same-day releases. By manually investigating such provider updates, we observed that they are prevalent when client and provider packages are hosted in the same codebase.

RQ3. How do client packages adopt same-day releases? We observed that the rate at which same-day releases are adopted by client packages is comparable with the adoption rate of regular releases. Yet, the speed at which same-day releases are adopted is significantly higher than the adoption speed of regular releases.

Based on our empirical investigation, we discuss implications for npm package developers to reason about the best practices regarding same-day releases. In specific, we discuss that the important trade-off between the value of a same-day release and its potential risks should be carefully evaluated by client packages, as well as how current automated dependency management tools can help in this sense. Also, the high rate of same-day releases suggests that popular packages should strive for optimized release pipelines. Finally, we discuss why provider packages should improve the awareness of how their releases impact client packages.

Our study presents substantial contributions in addition to prior studies about same-day releases. In summary:

- **We show that the number of same-day releases in npm is higher than in prior studied systems.** Packages in the npm ecosystem publish a significant larger number of same-day releases compared to other studied systems (Hassan et al., 2017; Lin et al., 2017). Therefore, our study is performed over a data set with characteristics different from prior studies.

- **We are the first to study the source code of same-day releases.** Our results rely on historical data from the source code of the deployed files by npm packages, providing novel information regarding the changes that are performed in same-day releases.

- **We show that same-day releases are particularly relevant in the npm ecosystem.** This is the first study to discuss the impact of same-day releases on the dependencies of a software ecosystem. In particular, we investigate how client packages adopt same-day releases of their providers.

5.2 Background & Operational Definitions

In this section we present background information and related work on same-day releases. We initially present related work to release management and same-day releases (Section 5.2.1). Next, we present our operational definition of same-day, prior-to-same-day, and regular releases (Section 5.2.2).

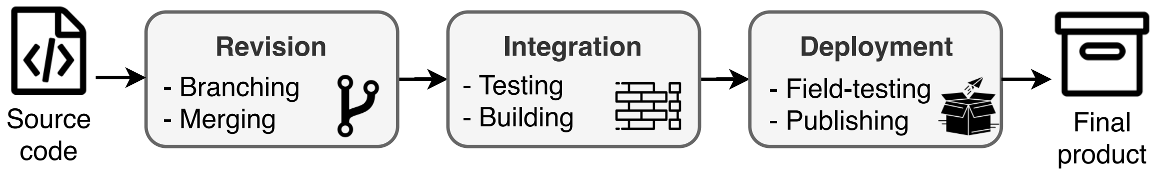


Figure 5.1: Simplified release management pipeline.

5.2.1 Release management

Release management is the set of processes that produces a final deliverable product from the source code. Software projects can adopt different activities for managing their releases (Mäntylä et al., 2015; Adams et al., 2015; Van der Hoek and Wolf, 2003; Erenkrantz, 2003) and complementary release management models were proposed in the literature (Kajko-Mattsson and Yulong, 2005; Adams and McIntosh, 2016). Figure 5.1 depicts a simplified release management pipeline. The simplified pipeline does not cover all possible phases and activities performed in the release management of a software project, but does explain the most common phases and activities that are relevant for the scope of our study. Essentially, our simplified pipeline synthesizes the release management models as proposed by Adams and McIntosh (2016), Lahtela and Jäntti (2011), Kajko-Mattsson and Yulong (2005), and Erenkrantz (2003). In the following, we describe each of the three phases in this pipeline.

Revision. In this phase, the source code that is generated by developers is committed to different active branches in the VCS and eventually merged into a production branch (the master branch). *Branching* is the process of managing different branches (or codelines) in the VCS, each one containing a version of the source code that complies with different levels of quality assurance. *Merging* is the process of combining the contents of a branch with the contents of another branch (e.g., merging a review with a master branch).

Integration. After a source code change is committed, a *build* process is started. In the build process, dependencies are linked and an executable artifact is generated. Although unit tests are likely run at development-time, automated integration processes also run a set of *tests*, including regression and integration tests.

Deployment. In the deployment phase, a release is *field-tested* and made available to the users. *Publishing* is the process of rolling out a release and making it available to the users. The published final product can be deployed and distributed to users by different means, such as package managers, web services, or mobile app stores.

Occasionally, a release needs to be expedited throughout the release management pipeline (Fox, 2002; Hamilton, 2007; Kerzazi and Adams, 2016; Hassan et al., 2017; Lin et al., 2017; Castelluccio et al., 2019). For example, Lahtela and Jäntti (2011) propose a model for release management that explicitly suggests publishing “urgent releases” when they are needed. In this scenario, developers might eventually publish a follow-up release in the same day of the previous release. Same-day releases indicate a new version that is developed and published within a restricted time-frame, regardless of the motivation behind expediting the release. The following interview excerpt illustrates the usage of same-day releases by the Mozilla development team (Adams et al., 2015):

> *"If there were a serious problem, like a huge number of crashes on a certain release (...) we would do a point release so that users wouldnt get the last release and would be automatically updated to the newer release (...) We call this a zero-day fix."*

Kerzazi and Adams (2016) studied the "botched releases" of a commercial web app. Botched releases are defined as releases that present a malfunction shortly after ("in a few minutes") being published. The authors identified that 72 out of 320 studied releases (22.5%) are botched—a non-negligible proportion of all releases. In a study about urgent releases in the Google Play Store[3] (Hassan et al., 2017), five different patterns of urgent releases were identified, of which three have a median time to repair of one day. Urgent releases were also studied on the Steam platform[4] for games distribution. Lin et al. (2017) found that 80% of the studied games publish at least one urgent release and that the major reasons for urgent releases are feature malfunctions, crashing, and visual bugs. Lin et al. (2017) also observed that the occurrence of same-day releases in the Steam platform is associated with urgent releases.

Same-day releases are especially important in software ecosystems because packages heavily depend on each other. Prior studies show that the interdependency of npm packages makes the release of a popular package to impact many other packages that directly or indirectly depend on the former (Kikas et al., 2017; Wittern et al., 2016; Decan et al., 2017). For example, Kikas et al. (2017) show that the removal of some npm packages can impact up to 30% of other packages in the ecosystem. Anecdotal evidence of this potential problem exists. For example, recently a failure in a popular

[3] `https://play.google.com/store/apps`
[4] `https://store.steampowered.com`

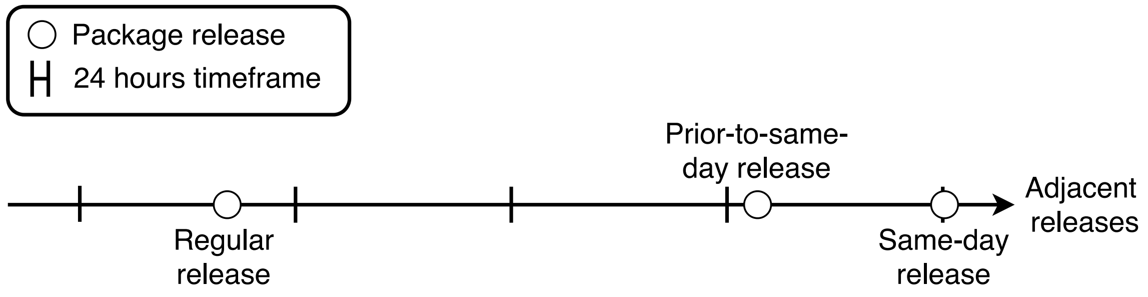


Figure 5.2: The release types that are investigated in our study.

package[5] with a single line of code caused a massive downtime on the npm ecosystem and a same-day release was published to solve the issue. Another well-known example is the "left-pad incident",[6] in which a package was removed from npm and led many packages in the ecosystem to fail. Within a few hours, a follow-up replacement package was published in npm and the ecosystem stability was recovered. In those two incidents, many package failures occurred because of transitive dependencies (i.e., a failure in a popular package propagated throughout the ecosystem), which denotes the risk brought by the dense interdependency of npm packages. Therefore, same-day releases of popular packages in large software ecosystems deserve proper attention. To the best of our knowledge, this is the first empirical study about same-day releases in a software ecosystem.

5.2.2 Same-day, prior-to-same-day, and regular releases

We classify adjacent releases into three types depending on the time at which they are published, namely same-day, prior-to-same-day, or regular release. Figure 5.2 illustrates the three release types.

[5]https://www.zdnet.com/article/another-one-line-npm-package-breaks-the-javascript-ecosystem/
[6]https://www.theregister.com/2016/03/23/npm_left_pad_chaos/

A *same-day release* is a package release that is published in npm in less than 24 hours after the previous release of the same package (according to npm's timezone). The predecessor of a same-day release is called a *prior-to-same-day release* (which is, by definition, published in npm less than 24 hours from the following same-day release). A prior-to-same-day release can also be a same-day release. For example, suppose that release r_2 of a package p is published in less than 24 hours after release r_1 and that release r_3 is published in less than 24 hours after release r_2. In this example, release r_2 of package p is both a same-day and a prior-to-same-day release. Throughout this paper, when a distinction between same-day and prior-to-same-day releases is needed (e.g., when we compare metrics for same-day with the same metrics for prior-to-same-day releases), we prioritize the classification of a release as same-day. We call releases that are published after another same-day release as *back-to-back same-day releases*. Considering our previous example, the release r_3 is a back-to-back same-day release (since it was published after the same-day release r_2). Finally, releases that are not same-day nor prior-to-same-day are classified as *regular releases*.

5.3 Data collection

Our general goal is to understand the prevalence, code changes, and adoption of the same-day releases that are published in the npm ecosystem. The reason for choosing this ecosystem is that it is the largest and most popular software ecosystem to date.[7] Consequently, the npm ecosystem is inherently relevant in practice and provides rich data for studying same-day releases.

[7] https://insights.stackoverflow.com/survey/2019

To accomplish our goal, we mine several pieces of data from the npm registry. Figure 5.3 depicts our data collection process. Four steps were employed in our data collection. The first step is performed to collect the `package.json` files from npm packages (Section 5.3.1). The second step selects packages for our study (Section 5.3.2). In the third step, we select the patch releases, identify same-day and prior-to-same-day releases, as well as identify dependencies in which the provider is updated (Section 5.3.3). Finally, the fourth step is to analyze how the deployed files change between adjacent npm releases (Section 5.3.4).

5.3.1 Collecting data from npm packages

In this step we collect meta-data about the packages that were published in npm between December 20, 2010 and May 13, 2019. We process the data as follows:

Obtain `package.json` files from npm packages. We fetch the `package.json` files of all published packages in the npm registry. The `package.json` file contains metadata about about all releases of a package, including the release version number and timestamp, as well as the adopted provider packages with their respective versioning statements. The fetched `package.json` files are limited to packages within the npm ecosystem and, therefore, do not include information about external applications that depend on an npm package.

Extract metadata from the `package.json` files of each package. We extracted the following information about the packages: the release version number, the release timestamp, the dependencies and the respective versioning statements that are used in each release. In total, we obtained the `package.json` file of 976,631 npm packages.

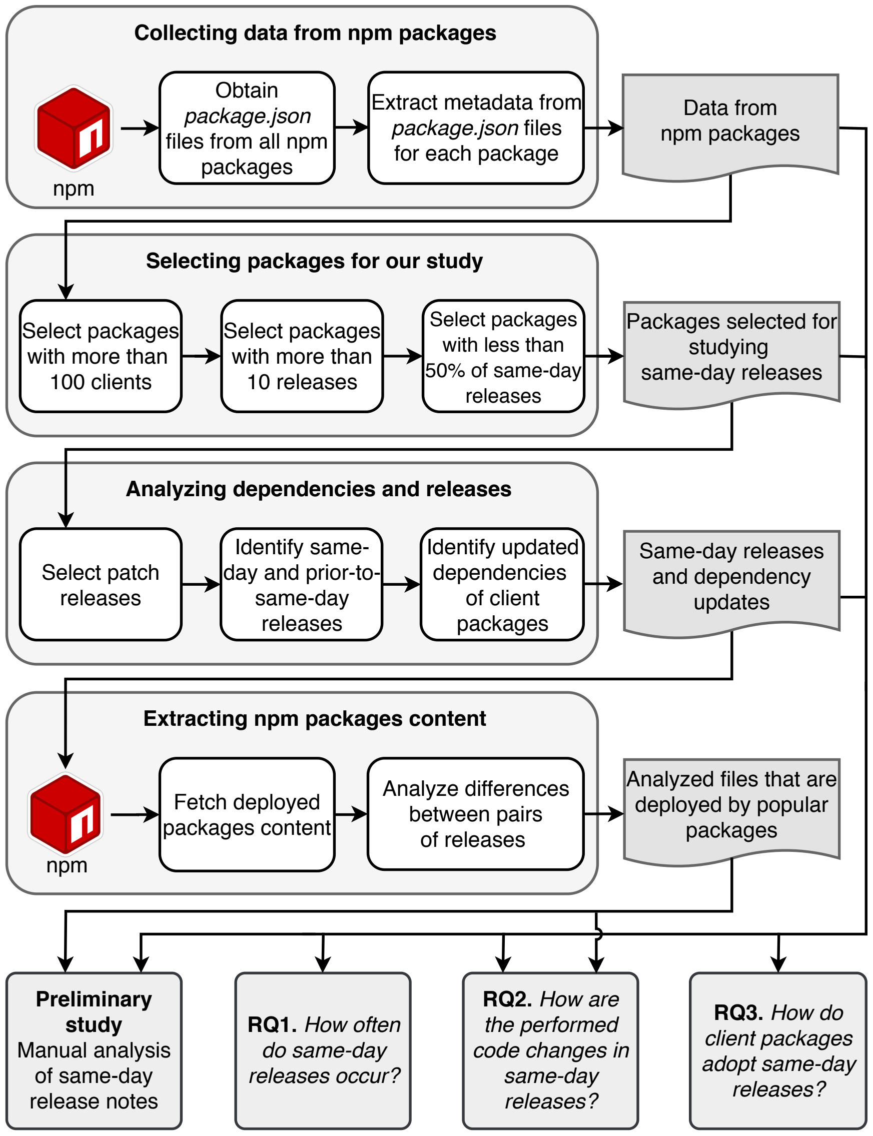

Collecting data from npm packages
npm
Obtain *package.json* files from all npm packages
Extract metadata from *package.json* files for each package
Data from npm packages
Selecting packages for our study
Select packages with more than 100 clients
Select packages with more than 10 releases
Select packages with less than 50% of same-day releases
Packages selected for studying same-day releases
Analyzing dependencies and releases
Select patch releases
Identify same-day and prior-to-same-day releases
Identify updated dependencies of client packages
Same-day releases and dependency updates
Extracting npm packages content
npm
Fetch deployed packages content
Analyze differences between pairs of releases
Analyzed files that are deployed by popular packages
Preliminary study Manual analysis of same-day release notes
RQ1. *How often do same-day releases occur?*
RQ2. *How are the performed code changes in same-day releases?*
RQ3. *How do client packages adopt same-day releases?*

Figure 5.3: Overview of our data collection process.

5.3.2 Selecting packages for our study

We are interested in understanding the same-day releases of popular packages in the npm ecosystem. Hence, in this step, we adopt a set of selection criteria that ensure that noise is filtered out and the phenomenon of interest is reliably captured. The following criteria were adopted:

Select packages with more than 100 clients. A specific goal of our research is to focus on releases that can potentially affect a large portion of the ecosystem as, for example, releases of a provider package that is adopted by many client packages. For this reason, we decide to select only releases of the popular npm packages, i.e., top-ranked packages by the number of client packages. Our definition of popular package is based on the number of client packages that a certain provider package has within the ecosystem. In particular, we study packages with more than 100 clients. This threshold was selected after inspecting the distribution of the total number of clients per provider package. We observed that provider packages with more than 100 clients lie in the outliers of this distribution and, therefore, are representative of provider packages that accumulate more client packages than the majority of the other provider packages in the ecosystem.

Select packages with more than 10 releases. This criterion aims at filtering out packages that rarely publish a release or that do not have a release history yet.

Select packages with less than 50% of same-day releases. We filter out packages that regularly publish same-day releases, since the development process adopted by those

Table 5.1: Summary of the number of releases, number of same-day releases, number of clients, and number of providers for the popular packages.

Popular packages	**Min.**	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
Number of patch releases	10	16	26	37.62	44	460
Number of same-day releases	0	3	6	9.89	12	138
Number of clients	100	158	306	2,034	928	128,999
Number of providers	1	10	18	26.29	34	200

packages inherently deals with releases developed within a 24 hours period (i.e., same-day releases are not unusual to them). In other words, same-day releases that are published as part of a regular schedule will not reflect the effects of time constraint that we are interested in observing.

The output of this step is a set of 1,893 packages. We refer to such packages as *popular packages*. Whenever we mention a same-day release in the remainder of our study, we refer to a release of a popular package. The clients and providers of the popular packages were also included in our data set, since in our study we analyze the number of providers of the packages that publish same-day releases, as well as how the clients of such packages react to their releases. By including the clients and providers of the popular packages, a total of 504,983 packages are kept in our data set. Table 5.1 describes statistics of the selected packages from which same-day releases are analyzed.

5.3.3 Analyzing dependencies and releases

We analyzed the dependencies and releases of the popular packages to generate data that could be used to answer our RQs. The data was generated by the following processes:

Select patch releases. Releases in npm can be developed in parallel branches. As a consequence, major and minor same-day releases can be identified (e.g., versions 1.0.1 and 2.0.0 being released on the same-day). However, in such cases, it is clear that the same-day release (2.0.0 in the previous example) is not a patch for the 1.0.1 release. To reduce such noisy same-day releases from our analysis, we selected only the patch releases (e.g., from 1.0.0 to 1.0.1) of each studied package.

Packages in npm can publish either patch, minor, and major releases. Moreover, such releases can be interleaved, for example, when a release 1.2.3 is published as a patch to release 1.2.2 while the release 2.0.0 is already published. The occurrence of interleaved releases can misrepresent same-day releases. For example, if releases are sorted by the version number, then the releases 1.0.1 and 2.0.0 are considered adjacent releases. If those two releases are published within 24 hours, the release 2.0.0 might be considered a same-day release. However, in this case, it is clear that 2.0.0 is not patching release 1.0.1. To deal with such versioning related challenges, we select only patch releases for our study.

Identify same-day and prior-to-same-day releases. We identified all releases that were published in less than 24 hours after the prior release. Also, we identified the releases that are prior-to-same-day or regular releases (see Section 5.2.2).

Identify updated dependencies of client packages. For each client package release, we identified the providers that were updated in that release. To determine the updates of providers, we resolved the version of each provider according to the versioning statement used by the client package in each release and the provider versions that were released up to the date of each client release (see Chapter 2). More formally, let $d_{c,p,r}$ represent a dependency from the client package c on the provider package p in release r of c. The resolved provider version is the latest version of p that satisfies the versioning statement used by c in r. Versioning statements are parsed using our own parser,[8] which fully adheres to the npm grammar.

The output of this step is a data set containing all the releases (with the identified patch same-day releases) and the dependencies of the client packages (with the identified updates) from the popular npm packages.

5.3.4 Extracting npm packages' contents

To understand the changes that are performed in same-day releases, we rely on data associated with the published source files by npm packages. When developers publish a release in npm, they use the `npm publish` tool,[9] which packs all public files of a package in a tarball and deploys the contents in npm. We obtain and analyze the deployed files of all releases of the popular npm packages.

Fetch deployed package contents. For each i-th release $r_{p,i}$ of a popular npm package p, we used the `npm pack` tool[10] to obtain its respective deployed tarball. This tarball contains all public files that are deployed in npm for a given release of a package.

[8] https://github.com/SAILResearch/replication-npm_downgrades
[9] https://docs.npmjs.com/cli/publish
[10] https://docs.npmjs.com/cli-commands/pack.html

Analyze differences between pairs of releases. After obtaining the tarballs, we extract their contents and analyze the difference between all files of two adjacent releases $\langle r_{p,i-1}, r_{p,i}\rangle$ of a package p. This analysis calculates the number of inserted and deleted lines, as well as the contents of the added and deleted lines.

The output of this step is the analysis of the differences between the deployed files of each pair of adjacent releases.

5.4 Preliminary study of same-day release notes content

In this section, we present our preliminary study on the contents of the same-day release notes. The objective of our preliminary study is to better understanding the notable changes that are reported in same-day releases.

Motivation. Same-day releases are occasionally published to address issues that impact client packages in an ecosystem. Reported incidents show that the open-source community is usually able to timely react to failures in one package that end up impacting a large portion of the ecosystem (e.g., a failure in a release of a provider package that is directly and indirectly adopted by many client packages). Also, prior research shows that addressing time-sensitive issues is a common development practice (Kerzazi and Adams, 2016; Lin et al., 2017; Hassan et al., 2017; Castelluccio et al., 2019; Adams et al., 2015). However, the characteristics of the changes that are introduced in same-day releases is still an open question. For example, it is not clear whether same-day releases in fact introduce important changes (e.g., bug fixes) or are simply a result of a routine task (e.g., merging a commit). Hence, in this preliminary study we characterize the changes that are introduced in same-day releases. Also, we investigate whether the proportion of same-day releases that have an associated release note is different from the proportion of prior-to-same-day and regular releases.

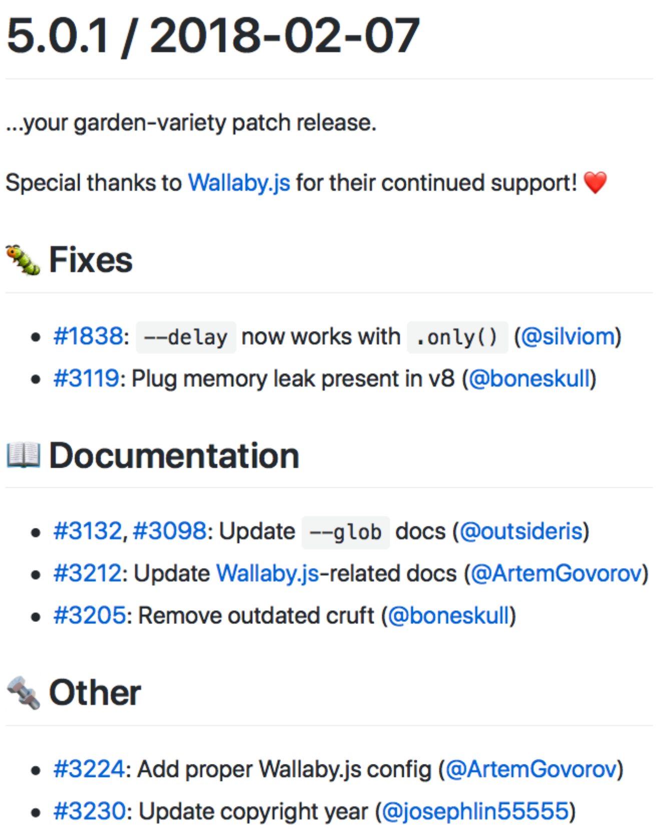
Approach. To understand the documented changes introduced in same-day releases, we perform an analysis of the same-day release notes. Release notes document the notable changes that are introduced in a new release compared to the previous release. Typically, a release note contains: 1) a list of notable changes grouped by the type of change (e.g., bug fixes, new features, documentation change, etc.), 2) the release version and the respective timestamp, and 3) links to Issue Tracking System (ITS) artifacts (such as issue reports, pull-requests, commits, or contributors). Some release notes do not include the release timestamp or any link to an ITS artifact, but all release notes have at least a list of notable changes and the respective release version. Figure 5.4 shows the release notes for release 5.0.1 of the popular mocha package.

As new releases of a package are published, the release notes are appended to a unique changelog file that is kept in the package's codebase repository and distributed with each release. Commonly, npm packages name their changelog files as "CHANGELOG.md". However, we also identified release notes in files that are named "HISTORY.md", "CHANGES.md", "RELEASE-NOTES.md", and "RELEASE.md", among other variations (e.g., files with HTML or txt extensions, or files with lower-case names). Hence, to identify changelog files of the popular npm packages, we search for files whose lower-cased name matches the following regular expression:[11]

(change* | history* | release*).(md | html | txt)

After identifying the changelog files, we estimate the proportion of same-day releases that have an associated release note. To establish a baseline, we also calculate

[11]In our regular expression notation, the | operator is a logical OR and the * operator is a non-greed zero-or-more matcher of any character. All other symbols are literal characters.


5.0.1 / 2018-02-07

...your garden-variety patch release.

Special thanks to Wallaby.js for their continued support! ❤️

🐛 Fixes

- #1838: `--delay` now works with `.only()` (@silviom)
- #3119: Plug memory leak present in v8 (@boneskull)

📖 Documentation

- #3132, #3098: Update `--glob` docs (@outsideris)
- #3212: Update Wallaby.js-related docs (@ArtemGovorov)
- #3205: Remove outdated cruft (@boneskull)

🔩 Other

- #3224: Add proper Wallaby.js config (@ArtemGovorov)
- #3230: Update copyright year (@josephlin55555)

Figure 5.4: Release note example of the popular package mocha.

the proportion of prior-to-same-day and regular releases that have an associated release note. We used the Fisher's Exact Test ($\alpha = 0.05$) with Holm-Bonferroni correction to determine whether there is any relation between the type of release (same-day, prior-to-same-day, and regular) and the publication of a release note. More formally, let SDR_p, PSDR_p, and REG_p be, respectively, the set of same-day, prior-to-same-day, and regular releases of a package p. We tested two null hypotheses: $H_0 : P_{\text{RN}}(\text{SDR}_p) = P_{\text{RN}}(\text{PSDR}_p), \forall p \in P$ and $H_0 : P_{\text{RN}}(\text{SDR}_p) = P_{\text{RN}}(\text{REG}_p), \forall p \in P$, where P_{RN} is the proportion of releases that have a release note. The magnitude of the differences (effect size) is calculated as the difference between the proportions.

Next, we proceed with a manual analysis of the same-day release notes (i.e., the release notes of same-day releases). We thoroughly read the same-day release notes' contents as well as the ITS artifacts that are linked in the release notes. During our reading process, we seek for deriving a detailed description of the notable changes of same-day releases. In particular, we carry out an open coding process (Stol et al., 2016), resulting in a set of categories that describe the typical changes that are introduced in same-day releases. During our open coding process, we achieved saturation after analyzing a random sample of 104 same-day release notes. Afterwards, we compare our categorization with a pre-existent categorization schema as proposed by Moreno et al. (2014). Therefore, in a next step, we also categorize the 104 same-day release notes with Moreno's categorization schema. The categories derived from our open coding process differs from the categories as proposed by Moreno et al. (2014) with respect to the level of detail. For example, the content of a release note can be categorized as a bug fix according to the categorization schema as proposed by Moreno et al. (2014), while our categorization denotes the type of change employed to fix the bug (e.g., changing the business logic) or, even, the type of bug that drove the change (e.g., fixing a memory leak).

We use an infographic (Lex et al., 2014) to visualize the occurrence and co-occurrence frequencies of the categories found in same-day release notes. In this infographic, the axis labelled "occurrence frequency" represents the total number of same-day release notes whose content was categorized with a certain category. In turn, the axis labelled "co-occurrence frequency" represents the number of same-day release notes of which the content was categorized into one, two, or three categories at once. The co-occurrent categories are identified by a dot matrix shown in the inforgraphic. A column-wise filled dots in the matrix represent the categories that co-occur in a same-day release note (exclusive categories appear as a single filled dot).

Observation 5.1) ***32% of the same-day releases have an associated release note.*** In turn, 34% of the prior-to-same-day releases and 37% of the regular releases have a release note. The difference between the proportion of same-day and prior-to-same-day, as well as between same-day and regular releases that have a release note is statistically significant (p-value < 0.05, Holm-Bonferroni corrected). Nonetheless, the largest difference is only 5% between same-day and regular releases. The negligible effect size shows that the existence of release notes does not seem to be related to the release type. Hence, we can conclude that the time constraint of same-day releases is not associated with publishing release notes. Nevertheless, the proportion of releases that have a release note is relatively low (approximately one third of all releases). The lack of release notes can be problematic for client packages, since they become unable to know which changes were introduced in a release and, therefore, cannot assess the impact of the introduced changes.

Observation 5.2) ***The analyzed same-day release notes typically refer to a fixed bug.*** In Figure 5.5, we summarize the categorization of the studied release notes according to the categorization schema as proposed by Moreno et al. (2014). The category "fixed bugs" occurred in more than 60 same-day release notes. Also, the content of 45 same-day release notes was categorized exclusively as "fixed bugs" (see the first column of the dot matrix), whereas the content of 4 same-day release notes was categorized with both "fixed bugs" and "changes to configuration files" categories (see the fifth column of the dot matrix). Despite the plurality of categories and co-occurrence of categories shown in Figure 5.5, we can observe that fixed bugs are more prevalent than any other category.

Observation 5.3) ***The most common changes in the studied same-day releases are to business logic, to correct the integration of a provider package, or to avoid an unrecoverable failure.*** In Table A.1 (Appendix A), we describe in details the 28 categories of changes that were introduced in same-day releases. While "changes in business logic" suggest that many changes introduced in same-day releases rely on the behaviour of the popular package itself (i.e., the package that publishes the same-day release), "correctly integrate provider" suggests that the changes introduced in same-day releases rely on the interaction between the popular package and its providers. Also, "avoid crashing" and "fix UI error" suggest that the changes introduced in same-day releases solve serious issues that affect client packages.

During our manual analysis, we observed that even routine tasks such as those categorized as "VCS chores" or "Fix typo" can address relevant issues. For example, release

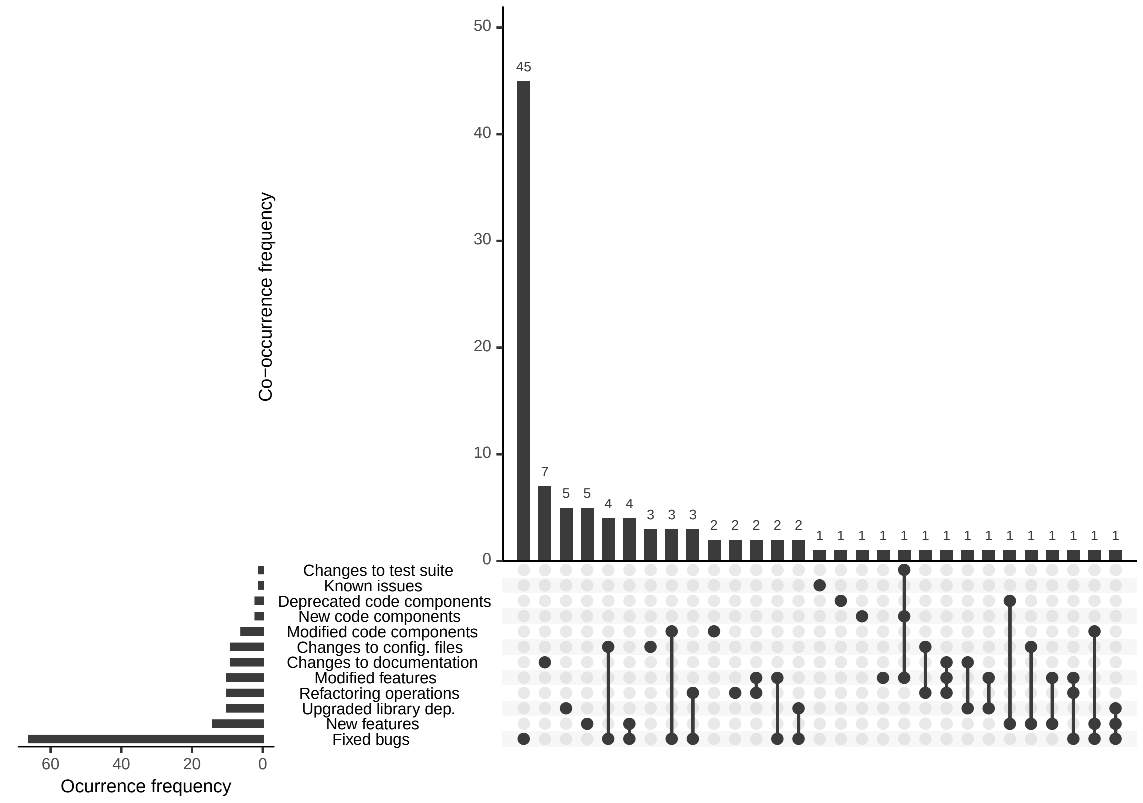


Figure 5.5: Summary of the categorization of notable changes in 104 same-day releases.

0.4.20 of the config package was published to address an issue caused by a non-merged commit. The following dialogue was captured in issue report #47 of the config package:

- "*Unfortunately this was the last commit using the old philosophy (commit to version branches vs. master), and it got missed.*"
- "*(...) Fix of _diffDeep function was critical for me.*"

Another issue report exemplifies simple yet serious issues that are addressed in same-day releases. A failure caused by an improperly placed comment in a CSS file was fixed in a same-day release:

- "*This CSS: (...) Produces this error (...) This is kicked because of the comment.*" (Issue report #163 of package postcss-import@8.0.2)

We also observed a same-day release note categorized as "Improve documentation" that admits the introduction of technical debt:

> "*Ideally the plugin would check for permissions and raise a more helpful error, but a note in the readme should help until then*" (Issue report #13037 of package gatsby@2.3.6)

Summary: A manual analysis of a random sample of same-day release notes reveals that simple yet important changes are introduced in same-day releases, primarily bug fixing changes.

5.5 Research questions

In this section, we present the motivation, approach, and results for each of our RQs.

5.5.1 RQ1: How often do same-day releases occur?

Motivation. The inherent time constraint of same-day releases can compel developers to expedite typical release activities. However, the extent to which development teams of packages in npm need to publish same-day releases is not understood. Quantifying the phenomenon of same-day releases is a first step towards determining how important it is for package development teams to adopt a suitable strategy to manage such releases.

Approach. To investigate how often same-day releases are published by popular npm packages, we first calculate the proportion of all releases that are same-day releases and the proportion of popular packages that publish at least one same-day release. More formally, let PP be the set of all popular packages, SDR_p the set of same-day releases of a package p, and R_p the set of all releases of a package p. Also, let the proportion of all releases of a package p that are same-day releases be denoted by $P_{\text{SD}}(\text{R}_p)$. We analyze the distribution of the following values:

$$P_{\text{SD}}(\text{R}_p) = \frac{|\text{SDR}_p|}{|\text{R}_p|}, \forall p \in \text{PP}$$

In turn, let the proportion of all popular packages that publish at least one same-day release be denoted by $P_{1\text{SD}}(\text{PP})$. We analyze the distribution of the following values:

$$P_{1\text{SD}}(\text{PP}) = \frac{|\{p : P_{\text{SD}}(\text{R}_p) > 0\}|}{|\text{PP}|}, \forall p \in \text{PP}$$

We also investigate whether some popular packages publish more same-day releases than others. To this end, we analyze a cumulative histogram showing the proportion of packages that publish a given proportion of same-day releases. We also calculate the proportion of back-to-back same-day releases (i.e., same-day releases followed by another), as well as the proportion of packages that publish back-to-back same-day releases at least once.

We check whether there is an association between the number of same-day releases and the number of overall releases of the popular packages. We calculate the Spearman's ρ measure of correlation between those two variables. The choice of a rank-based measure of correlation is due to the skewness and the presence of outliers in both variables. Also, we interpret the correlation measure according to the following classification (Schober et al., 2018):

$$\text{Correlation} = \begin{cases} \text{negligible}, & \text{if } 0 \leq |\rho| \leq 0.1 \\ \text{weak}, & \text{if } 0.1 < |\rho| \leq 0.39 \\ \text{moderate}, & \text{if } 0.39 < |\rho| \leq 0.69 \\ \text{strong}, & \text{if } 0.69 < |\rho| \leq 0.89 \\ \text{very strong}, & \text{if } 0.89 < |\rho| \leq 1 \end{cases}$$

Observation 5.4) ***The large majority of the popular packages (96%) published at least one same-day release.*** Also, 26% of the studied releases are same-day releases. Based on this result, developers of a popular package can expect high odds of a release being followed by a same-day release, suggesting that activities aimed at managing such

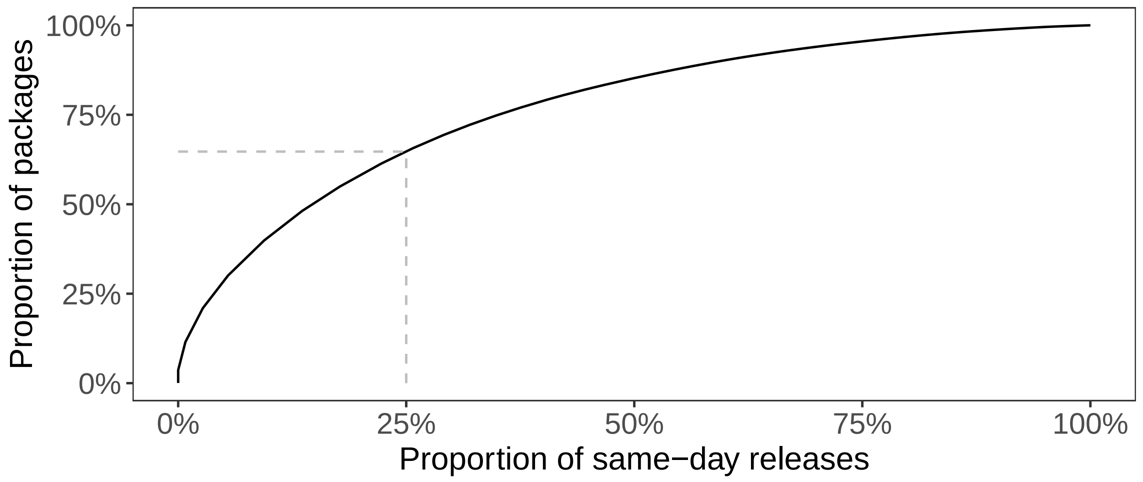


Figure 5.6: Cumulative histogram for the proportion of published same-day releases per package.

same-day releases (e.g., having an optimized pipeline for same-day releases) can be beneficial to popular packages.

Observation 5.5) ***Same-day releases are more prominent for a particular set of popular packages.*** Figure 5.6 depicts the proportion of packages that publish a given proportion of same-day releases. We can observe that the number of same-day releases per package is fairly skewed. In particular, 25% of all studied popular packages publish 65% of all studied same-day releases.

Observation 5.6) ***Same-day releases are commonly followed by another same-day release.*** 39% of the same-day releases are back-to-back and 70% of the popular packages publish at least one back-to-back same-day release. This observation shows that developers should strive for adopting rigorous and optimized process to deal with same-day releases. Also, a small set of the popular packages (25%) publish a large proportion of the back-to-back same-day releases (76%).

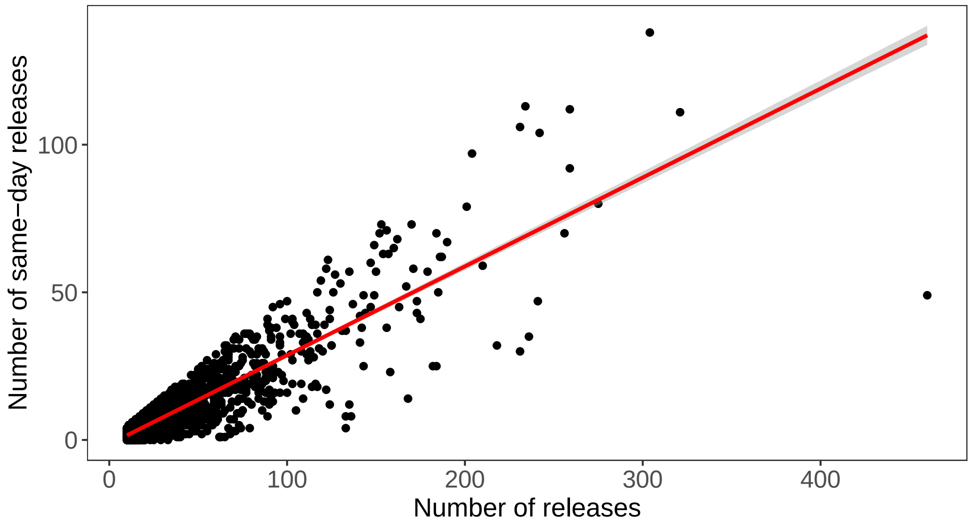


Figure 5.7: The relation between the number of same-day and the number of total releases of each popular package.

Observation 5.7) ***The number of same-day releases and the number of releases of the popular packages are strongly correlated.*** For the majority of the popular packages, as the number of releases of a package increases, the number of same-day releases also increases. The relation between the number of releases and the number of same-day releases is shown in Figure 5.7. There is a strong monotonic relation between the number of releases and the number of same-day releases of a package (Spearman's $\rho = 0.73$, p-value < 0.05). The monotonic relation between the number of releases and the number of back-to-back same-day releases is also strong (Spearman's $\rho = 0.85$, p-value < 0.05).

RQ1: How often do same-day releases occur?

- Same-day releases are more than one-quarter (26%) of all releases of a popular package.
- 25% of all popular packages publish 65% of all same-day releases.
- More than one-third (39%) of the same-day releases are followed by another same-day release.

5.5.2 RQ2: How are the performed code changes in same-day releases?

In this section, we describe the differences between the code changes performed in same-day and other release types. We focus on two main aspects of the code changes. In Section 5.5.2.1, we study a set of *change metrics* for same-day releases. In Section 5.5.2.2, we study the *modified file types* in same-day releases. We adopt the performed changes in other release types (prior-to-same-day and regular releases) as a baseline for studying the performed changes in same-day releases.

5.5.2.1 Change metrics

Motivation. Change metrics are generally associated with software quality (Graves et al., 2000; Nagappan and Ball, 2005; Moser et al., 2008) and maintainability (Eski and Buzluca, 2011). The larger the changes performed in a release, the larger the probability of introducing issues. The analysis of change metrics can reveal the extent to which the development of same-day releases is more prone to introducing issues than other release types.

Approach. We calculate and compare a set of change metrics for same-day and other release types. The following metrics were calculated for all popular package releases:

Number of modified lines of code (mLOC): We calculate how many lines of code are modified (inserted or deleted) in the files that are deployed on npm as part of a package's release (see Section 5.3.4 for a description of how such files are obtained). More formally, for each i-th release $r_{p,i}$ of a package p, $\text{mLOC}_{r_{p,i}}$ is the number of modified lines of code in the files associated with $r_{p,i}$ (in comparison with the previous release $r_{p,i-1}$).

Number of modified source code files: The analysis of the number of modified source code files in a release reveals how much change occurs in source code files instead of other files related to, for example, configuration or documentation. Hence, this metric measures the extent to which a change introduces modifications in the behaviour of a package. The behaviour of an npm package is embedded into JavaScript source code files. However, two other popular languages in npm that are compiled to JavaScript (and, therefore also embed packages behaviour) are TypeScript and CoffeeScript. We identify source code files by using a regular expression that captures file names with the extension "`.js`" (for JavaScript), "`.ts`" (for TypeScript), or "`.coffee`" (for CoffeeScript) within a "test", "src", "lib" or root directory.

Number of updated providers: Updating a provider package requires a migration effort that, in many cases, involves understanding the implications of the introduced changes in the new provider version. Understanding such implications can require time from developers, which could be scarce for same-day releases.

Thus, this metric can help understand the extent to which developers are dedicating their effort to update providers in a release. To calculate this metric, we sum the number of provider packages that are updated in a release. To detect whether a provider package p is updated in a release $r_{c,i}$ of a client package c, we resolve the version of p that is used by c in the releases $r_{c,i-1}$ and $r_{c,i}$. If the resolved version of p in $r_{c,i-1}$ is smaller than the resolved version of p in $r_{c,i}$, then an update of p is accounted for in $r_{c,i}$.

We compare the value for the aforementioned metrics between different release types of the popular packages. Our objective is to perform pair-wise comparisons between same-day releases and the other release types of one package. To perform such paired comparisons, we calculate the distribution of each metric per release (for all same-day and prior-to-same-day releases) and the distribution of the median of each metric per package (for same-day and prior-to-same-day releases of all packages). More formally, let PP be the set of all popular packages. Also, let SDR_p, PSDR_p, and REG_p be, respectively, the set of same-day, prior-to-same-day, and regular releases of a package p. Now let $m(r_{p,i})$ denote the calculation of a metric m at the i-th release $r_{p,i}$ of a package p. To compare same-day releases with prior-to-same-day releases for a metric m, we use Wilcoxon Signed-rank ($\alpha = 0.05$) (Bauer, 1972) to test the following null hypothesis (for all popular package $p \in \text{PP}$):

$$H_0 : m(r_{p,i}) = m(r_{p,i-1}), \forall i : r_{p,i} \in \text{SDR}_p \wedge r_{p,i-1} \in \text{PSDR}_p$$

In addition, to compare same-day releases with regular releases for a metric m, we use Wilcoxon Signed-rank ($\alpha = 0.05$) to test the following null hypothesis (for all popular package $p \in \mathrm{PP}$):

$$H_0 : Median(m(r_{p,i})) = Median(m(r_{p,j})), \forall i, j : r_{p,i} \in \mathrm{SDR}_p \land r_{p,j} \in \mathrm{REG}_p$$

The calculated p-values are corrected with the Holm method for multiple comparisons (same-day vs. prior-to-same-day and regular releases). The magnitude of the difference between the distributions is assessed using the Cliff's Delta d estimator of effect size (Cliff, 1996). We interpret the value of d according to the following thresholds (Romano et al., 2006):

$$\text{Effect size} = \begin{cases} \text{negligible,} & \text{if } 0 \leq |d| \leq 0.147 \\ \text{small,} & \text{if } 0.147 < |d| \leq 0.33 \\ \text{medium,} & \text{if } 0.33 < |d| \leq 0.474 \\ \text{large,} & \text{if } 0.474 < |d| \leq 1 \end{cases}$$

For each metric m whose difference across release types has a non-negligible effect size, we visualize the distribution of two ratios: *same-day to prior-to-same-day for a metric m* and the *same-day to regular for the median of a metric m.* Letting $m(r_{p,i})$ denote the calculation of a metric m at the i-th release $r_{p,i}$ of a package p, the ratios are defined as follows:

Same-day to prior-to-same-day ratio for a metric m: The same-day to prior-to-same-day ratio $SP(r_{p,i}, m)$ for a metric m at a same-day release $r_{p,i}$ is:

$$SP(r_{p,i}, m) = \frac{m(r_{p,i})}{m(r_{p,i-1})}, \forall i : r_{p,i} \in \mathrm{SDR}_p \land r_{p,i-1} \in \mathrm{PSDR}_p$$

Same-day to regular ratio for the median of a metric m: The same-day to regular ratio $SR(p, m)$ for the metric m of a package p is given by:

$$SR(p, m) = \frac{Median(m(r_{p,i}))}{Median(m(r_{p,j}))}, \forall i, j : r_{p,i} \in \mathrm{SDR}_p \land r_{p,j} \in \mathrm{REG}_p$$

Observation 5.8) ***In 32% of the same-day releases, the number of modified lines of code is larger than in their associated prior-to-same-day releases.*** Also, for 24% of the popular packages, the median number of modified lines of code (median mLOC) is larger in same-day releases than in regular releases. Table 5.2 shows that the median mLOC for same-day, prior-to-same-day, and regular releases is respectively 14, 34, and 32. In addition, Tables 5.3 and 5.4 show that the difference of the number of mLOC between same-day releases and the other release types is statistically significant (p-value < 0.05, Holm-corrected), with a small ($|d| = 0.323$) and medium ($|d| = 0.366$) effect sizes for prior-to-same-day and regular releases, respectively. The distributions of the same-day to prior-to-same-day ratio and the same-day to regular ratio for the number of modified lines of code are shown in Figure 5.8. We also observe that the difference between the number of modified source code files in same-day releases

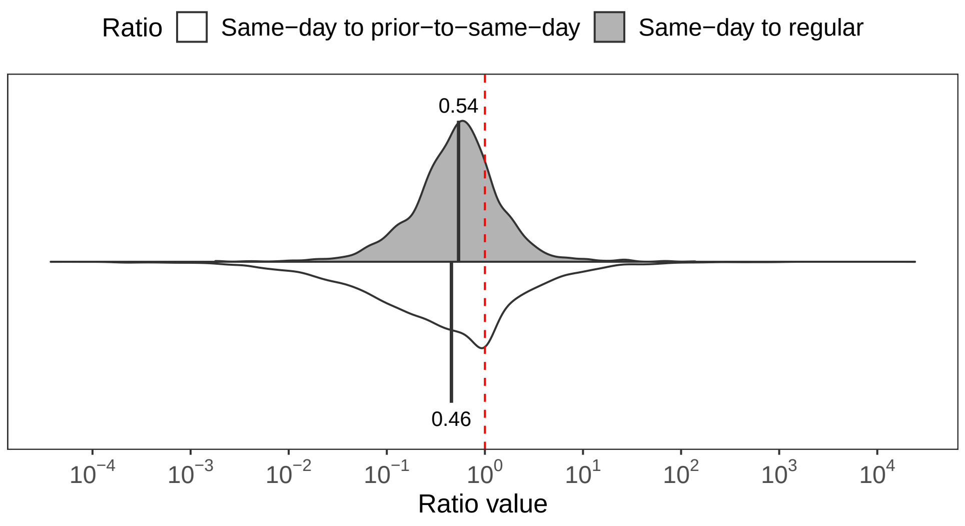


Figure 5.8: Same-day to prior-to-same-day and same-day to regular ratios for the mLOC metric. The red dotted line indicates a ratio of 1.

Table 5.2: Summary of the mLOC metric for same-day, prior-to-same-day, and regular releases.

Release type	**Min.**	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
Same-day	2.0	6.0	14.0	243.2	42.0	178,314.0
Prior-to-same-day	1.0	11.0	34.0	544.4	126.0	483,987.0
Regular	2.0	11.0	32.0	602.1	117.0	679,894.0

and other release types is statistically significant with a negligible effect size (see Tables 5.3 and 5.4). For the number of modified source code files, both the median same-day to regular ratio and the median same-day to regular ratio is equal to 1.

Observation 5.9) ***Despite the short time window, 17% of the same-day releases update at least one provider.*** In turn, 62% of the prior-to-same-day releases have at least one provider updated. The difference between the distributions of the number of

Table 5.3: Statistical difference between the change metrics for same-day releases vs. prior-to-same-day releases.

Metric	**Significant?**	**Effect size**
Number of modified lines of code	Yes	Small ($d = -0.323$)
Number of modified source code files	Yes	Negligible ($d = -0.142$)
Number of updated providers	Yes	Large ($d = -0.521$)

Table 5.4: Statistical difference between the change metrics for same-day releases vs. regular releases.

Metric	**Significant?**	**Effect size**
Median of modified lines of code	Yes	Medium ($d = -0.366$)
Median of modified source code files	Yes	Negligible ($d = -0.138$)
Median number of updated providers	Yes	Large ($d = -0.749$)

provider updates in same-day releases and the other release types is statistically significant and the effect size is large (see Tables 5.3 and 5.4). The distributions of the same-day to prior-to-same-day ratio and the same-day to regular ratio for the number of updated providers are shown in Figure 5.9. Also, we observe that 18% of the providers that were updated in the prior-to-same-day release are the same as the providers that were updated in the same-day release. By manually analyzing such cases, we found that they are predominant in packages that are maintained in monolithic repositories (Jaspan et al., 2018) (i.e., a single repository that hosts multiple packages), as for example the popular angular[12], babel[13], and react[14] packages. Such packages are part of a main project that maintains several independent packages in the same repository. Each independent package is used as dependency by other packages in the main project (in

[12]https://github.com/angular/angular/tree/master/modules
[13]https://github.com/babel/babel/tree/master/packages
[14]https://github.com/facebook/react/tree/master/packages

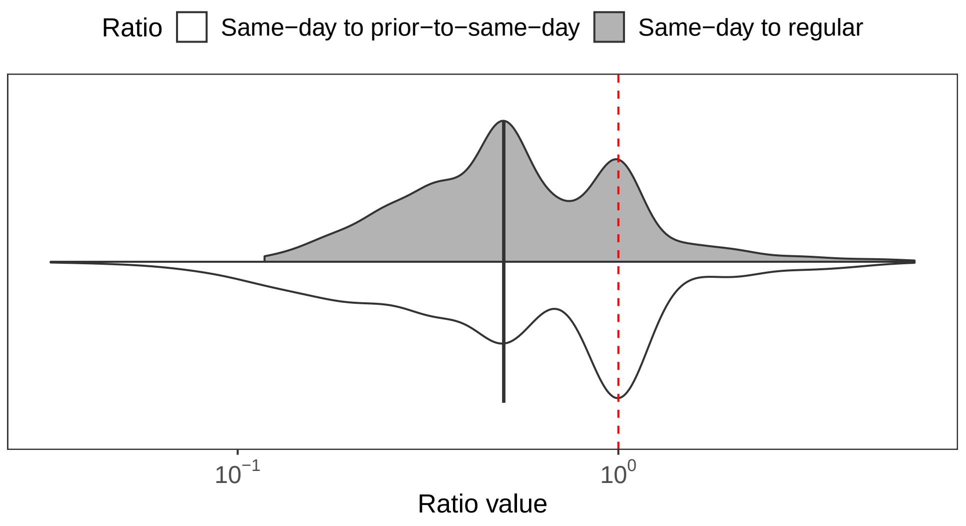


Figure 5.9: Same-day to prior-to-same-day and same-day to regular ratios for the number of updated providers. The red dotted line indicates a ratio of 1.

turn, such packages can be reused by other packages in npm). This result suggests that, although packages pertaining to the same main project are independently maintained, there is an association between the releases of such packages that can drive same-day releases. For example, perhaps the changes in a provider package propagates to client packages that are maintained in the same monolithic repository. In this case, client packages should consider refactoring to avoid such a change propagation.

5.5.2.2 Modified file type

Motivation. In Section 5.5.2.1, we analyzed how the values of change metrics from same-day releases compare those to those from prior-to-same-day and regular releases. However, changes can be performed in different portions of a software package. For example, releases that modify the source code (e.g., files that control the package's business logic) are likely to be either more important or impactful to client packages than

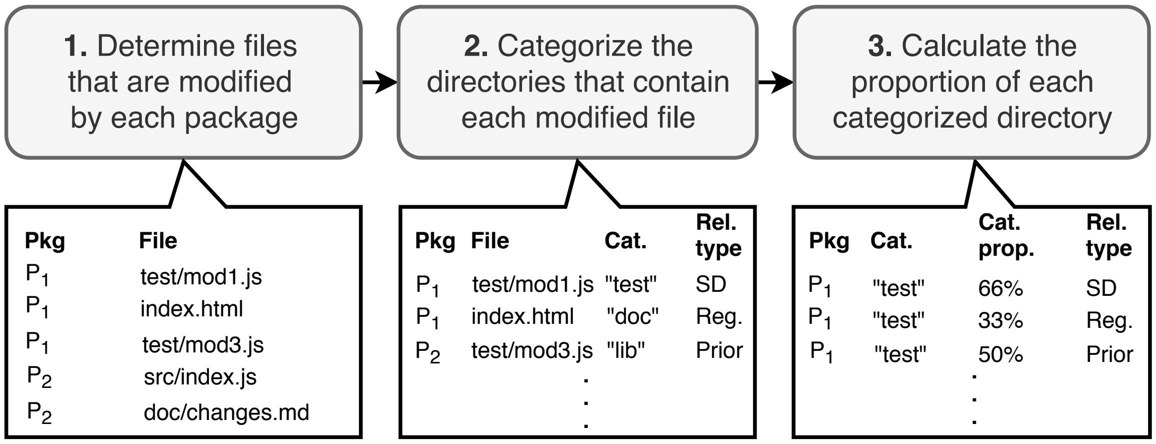

1. Determine files that are modified by each package
2. Categorize the directories that contain each modified file
3. Calculate the proportion of each categorized directory
Pkg File
P1 test/mod1.js
P1 index.html
P1 test/mod3.js
P2 src/index.js
P2 doc/changes.md
Pkg File Cat. Rel. type
P1 test/mod1.js "test" SD
P1 index.html "doc" Reg.
P2 test/mod3.js "lib" Prior
Pkg Cat. Cat. prop. Rel. type
P1 "test" 66% SD
P1 "test" 33% Reg.
P1 "test" 50% Prior

Figure 5.10: Our approach to study the modified file types in same-day and other release types.

those releases that modify documentation-related files. In this RQ, we want to determine whether the time constraint of same-day releases has any effect on the locations of changes. By answering this RQ, we provide complementary evidence regarding the source code changes that are performed in same-day releases, beyond the scope of the change metric measurement performed in Section 5.5.2.1.

Approach. Specific file types are stored in specific directories within the directory structure of npm packages. For example, a folder named `test` is likely to store test-related files. We studied the structure of the directories that store the modified files in same-day and other release types. Figure 5.10 depicts our approach:

1. *Determine files that are modified by each package:* To determine which files are modified in each release of a popular package, we rely on the deployed files that are published with each package release (see Section 5.3.4). For each deployed

file of a popular package p, we calculate the differences (per line) in the file contents between release $r_{p,i-1}$ and release $r_{p,i}$ (considering that release $r_{p,i-1}$ is adjacent to release $r_{p,i}$). We consider that a file is modified between two adjacent releases whether the difference of the content is non-empty.

2. *Categorize the directories that contain each modified file:* To categorize the directories that contain each modified file, we initially manually inspect the directory structure of a representative random sample (95% C.L., ±5% C.I.) of the popular npm packages. As a result of this manual inspection, we describe a set of directory types that are typically found in the directory structure of popular npm packages. Each directory type clearly separates files by their role (e.g., test-related files are stored in a different directory than documentation-related files). We also derive a set of regular expressions that are able to identify each directory type within the directory structure of a package. After identifying each directory type, we performed a sanity check to determine the accuracy of our method to categorize the packages' directories. In our sanity check, we verify whether the directories were correctly categorized according to the regular expressions. We observed that our regular expressions are able to accurately identify the directory types. For the sake of simplicity, we refer to files that are contained in a certain directory category as related files to that category. For example, we refer to files that are contained in a directory categorized as "test" as "test-related files". We used regular expressions to categorize the directory structure according to the following categories:

test Directory containing test-related files. Any directory whose name contains the string "test", "tests", "spec", or "specs" is deemed as a test directory.

lib Directory containing library-related files. Any directory whose name contains the string “lib” or “libs” is deemed as a library directory.

src Directory containing production-related files. Any directory whose name contains the string “src” is deemed as a production directory.

examples Directory with files containing examples of the package’s usage. Any directory whose name contains the string “example” or “examples” is deemed as an example directory.

doc Directory containing package’s documentation. Any directory whose name contains the string “doc*” (such as “doc”, “docs” or “documentation’) is deemed as a documentation directory.

build Directory containing build-related files. Any directory whose name contains the string “build” or “ci” (for “continuous integration”) is deemed as a build directory.

others Directories that do not fit in the prior categories, or contain files from a wide variety of file types, including the root directory.

3. *Calculate the proportion of each categorized directory:* For each release type, we calculate the proportion of releases that modify files that are stored in each directory category. More formally, we define $P_{d_c}(\text{SDR}_p)$, $P_{d_c}(\text{PSDR}_p)$, and $P_{d_c}(\text{REG}_p)$ as the proportion of same-day, prior-to-same-day, and regular releases of a package p that modifies a file in a directory categorized as d_c, respectively. For example, a package p_1 can modify test-related files in 66% of its same-day releases. Similarly,

p_1 can modify test-related files in 33% and 50% of its regular and prior-to-same-day releases, respectively. In this case, $P_{\text{“test”}}(\text{SDR}_{p_1}) = 0.66$, $P_{\text{“test”}}(\text{PSDR}_{p_1}) = 0.33$, and $P_{\text{“test”}}(\text{REG}_{p_1}) = 0.5$.

We use the Wilcoxon Signed Rank ($\alpha = 0.05$) to test the null hypothesis that the proportion of same-day releases that modify files in a certain directory category is equal to the proportion in prior-to-same-day and regular releases. The comparisons were paired per package, i.e., the proportion of same-day releases that modify a certain file type is compared with the proportion of prior-to-same-day and regular releases of the same package that modify that file type. For example, for the directory category “test”, we test the null hypotheses $H_0 : P_{\text{“test”}}(\text{SDR}_p) = P_{\text{“test”}}(\text{PSDR}_p)$, where $P_{\text{“test”}}(\text{SDR}_p)$ and $P_{\text{“test”}}(\text{PSDR}_p)$ are, respectively, the proportion of test-related files that are modified in SDR_p (the set of all same-day releases of a package p) and PSDR_p (the set of all prior-to-same-day releases of a package p). We use the Holm correction method to correct the obtained p-values for multiple comparisons (across combinations of directory categories and release types).

Observation 5.10) ***The proportion of modified file types between same-day releases and other release types are statistically significant but the effect sizes are negligible.*** Despite the difference in change size between same-day releases and other release types (see Section 5.5.2.1), the type of files that are modified are essentially the same regardless of the release type. In particular, we observe that, for all directory categories, with the exception of the “doc” directory for same-day vs. prior-to-same-day releases (p-value = 0.334, Holm corrected), the differences are all statistically significant. However, for the statistically significant differences, the effect size is negligible. This observation reinforces the importance of same-day releases since, in general, they modify the same amount of each file type compared with prior-to-same-day and regular releases.

RQ2: How are the performed code changes in same-day releases?

- 32% of the same-day release modify more lines of code than their associated prior-to-same-day releases.
- 17% of the same-day releases update at least one provider. Such updates are predominant for provider packages that are maintained in monolithic repository as the client package.
- Although the difference on the proportion of modified file types between same-day releases and other release types is statistically significant, the effect sizes are all negligible.

5.5.3 RQ3: How do client packages adopt same-day releases?

Motivation. In a software ecosystem environment, same-day releases have an important role, since problematic releases of a popular package can cause issues in a large proportion of its client packages. Development teams that need to publish a same-day release can benefit from knowing how client packages adopt such releases. Therefore, same-day releases can target the right set of client packages and the adoption of such releases can be maximized. In fact, a prior study that surveyed software ecosystem developers shows that two-thirds of the provider developers are interested in knowing whether their client packages adopt the latest release (Haenni et al., 2014). Therefore, it is important to understand the frequency and the speed with which client packages adopt same-day releases.

Approach. In this RQ, we study two aspects regarding the adoption of same-day releases by client packages, namely the *adoption frequency* and the *adoption lag*. To establish a baseline, we also study the adoption frequency and the adoption lag of prior-to-same-day and regular releases.

Adoption Frequency: To study the adoption frequency, we calculate the proportion of same-day, prior-to-same-day, and regular releases of the popular provider packages that are explicitly adopted by a client package. In our adoption frequency analysis, we focus on explicit adoption only. Explicit adoptions require the versioning statement to be changed by the client package (see Chapter 2). An explicit adoption represents a deliberate decision of the client to update the provider to a newer version. We use Fisher's Exact Test with Holm-Bonferroni correction to determine whether the difference in the proportion of same-day releases that are explicitly adopted is independent of the proportion of prior-to-same-day and regular releases. In addition, we study the proportion of adoptions of prior-to-same-day releases that occur when the same-day release is already published. Our objective with this latter analysis is to determine whether provider packages need to improve the communication of same-day releases to client packages.

Adoption Lag: We study the adoption lag of same-day and regular releases. The adoption lag $l_{p,c,i}$ is defined as the period spanned between the adoption of a release $r_{p,i}$ of the provider package p and the explicit update from the release $r_{p,i-1}$ to the release $r_{p,i}$ by a client package c, where $r_{p,i-1}$ is the provider release that precedes $r_{p,i}$ (see Figure 5.11). In essence, the adoption lag measures the length of the period during which a client package remains using the previous provider's release after the next provider release is published. If the introduced changes

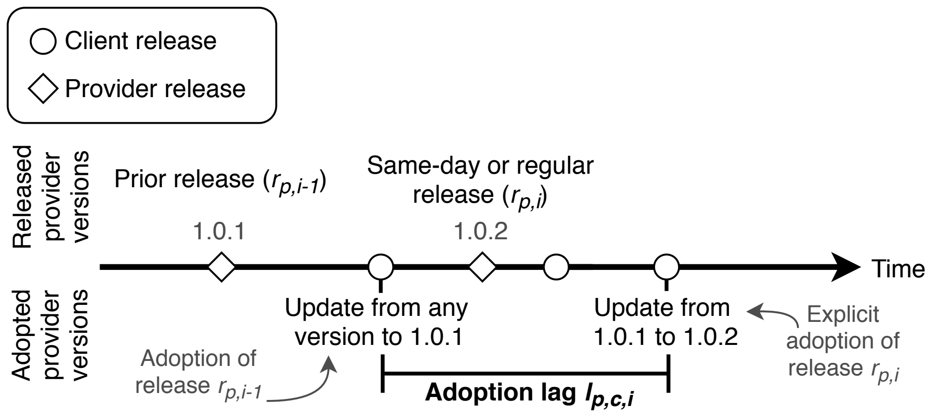

○ Client release
◇ Provider release
Released provider versions
Prior release ($r_{p,i-1}$)
1.0.1
Same-day or regular release ($r_{p,i}$)
1.0.2
Time
Adopted provider versions
Update from any version to 1.0.1
Update from 1.0.1 to 1.0.2
Explicit adoption of release $r_{p,i}$
Adoption of release $r_{p,i-1}$
Adoption lag $l_{p,c,i}$

Figure 5.11: Adoption lag of a same-day release.

in same-day releases are important to client packages, we should observe a fast adoption of such releases. The reason for accounting for explicit adoption only is that implicit adoptions occur automatically right after the provider release is published. Also, explicit adoption is a well-thought action from the client package, which suggests that client packages have a clear interest in the explicitly adopted provider release. For example, supposing a client package c that explicitly adopts version 1.2.2 of a provider package p, the following sequence of actions needs to be performed for an explicit update:

1. Provider p publishes a release in npm (e.g., version p@1.2.3)
2. Client package c updates the versioning statement (e.g. from "p": "`1.2.2`" to "p": "`1.2.3`")
3. Client package c publishes a new release in npm.

We measure the adoption lag in terms of the number of hours elapsed (time lag) and the number of client releases (release lag). For example, we can either measure the elapsed time between the release $r_{p,i}$ and the adoption of $r_{p,i}$ or, alternatively, we can measure the number of client releases that are published in the meantime between the release $r_{p,i}$ and the adoption of $r_{p,i}$. In addition, the release lag is measured in terms of major, minor, and patch releases of the client package (see Chapter 2). In our calculation of major, minor, and patch releases, we account only for the higher-level releases (major, minor, and patch). For example, if the client package publishes the release 1.2.3, 1.2.4 (patch), and 1.3.0 (minor) before the explicit provider update, we only account for the higher-level minor release (1.2.4 to 1.3.0). To evaluate the difference between the adoption lag of same-day and regular releases, we use the Wilcoxon Signed-rank test ($\alpha = 0.05$). We assessed the magnitude of the difference with the Cliff's Delta estimator of effect size.

Observation 5.11) ***The explicit adoption rate of same-day releases is 18% lower than that of regular releases.*** While 39% of the same-day releases are explicitly adopted by a client package, 30% of the prior-to-same-day and 57% of the regular releases are explicitly adopted. The differences are both statistically significant (p-value < 0.05 for Fisher's Test, Holm-Bonferroni corrected). Moreover, we observe that client packages might be inadvertently adopting the prior-to-same-day when, in fact, the same-day release is already published. In total, 14% of the explicit adoptions of a prior-to-same-day release occurs when the same-day release is already published, suggesting a lack of communication regarding the changes and importance of same-day releases. As the difference in the number of same-day, prior-to-same-day, and regular releases that have a release note is at most 5%, we conjecture whether popular packages use other means to signal to client packages that a same-day release is available. In particular, we investigate whether prior-to-same-day releases are annotated with a deprecation message. We observed that only 3% of the prior-to-same-day releases that are adopted when a same-day release is already published are flagged as deprecated. In fact, a closer look reveals that only 7% of all prior-to-same-day releases are ever flagged as deprecated. We thus conclude that development teams of popular packages should make an effort to signal to client packages that there is a same-day release that addresses issues in the prior-to-same-day release (e.g., by improving release notes and deprecating the prior-to-same-day releases)

Observation 5.12) ***Same-day releases are explicitly adopted faster than regular releases.*** The median adoption time lag for same-day releases (9 hours) is approximately 25 times smaller than the median for regular releases (224 hours). Figure 5.12 shows the distribution of the time lag for same-day and regular releases and Table 5.5 summarizes its values. The difference in the distribution between same-day and regular releases is statistically significant (p-value < 0.05, Holm-Bonferroni corrected) and the effect size is large ($|d| = 0.626$). We also observed that 61% of the explicit adoption of same-day releases occurs in less than 24 hours, which shows that many explicit adoptions of same-day releases are performed relatively fast in comparison to regular releases (the overall median explicit adoption time lag is 1,102 hours). We conjecture that such fast adoptions are driven by the bug fixes that are incorporated in same-day releases (see Section 5.4).

In terms of release lag to adopt same-day or regular releases, the difference between same-day and regular releases is not statistically significant for patch releases (p-value < 0.05, Holm-Bonferroni corrected). For minor and major releases, the differences are statistically significant (p-value < 0.05, Holm-Bonferroni corrected) and the effect sizes are negligible ($|d| = 0.055$) and small ($|d| = 0.192$), respectively. The median adoption lag for both same-day and regular releases is one release (see Table 5.6).

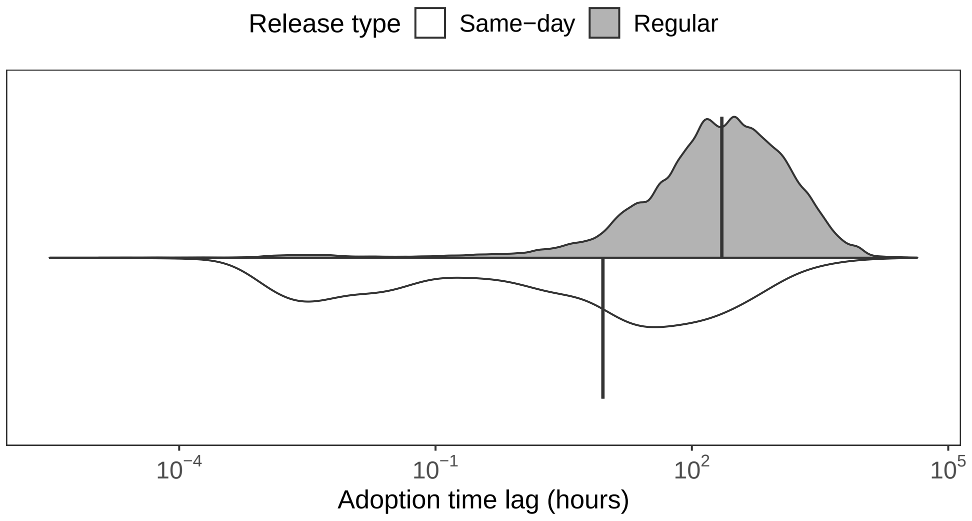


Figure 5.12: Adoption lag for same-day releases.

Table 5.5: Summary of the adoption time lag for same-day and regular releases.

Release type	**Min.**	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
Same-day	< 0.01	0.03	8.77	173.08	76.75	33,684.56
Regular	< 0.01	65.20	224.40	693.70	714.50	43,546.10

Table 5.6: Summary of the number of patch, minor, and major releases to explicitly adopt a same-day and a regular release.

Release level (client)	**Release type (provider)**	**Min.**	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
Patch	Same-day	1.0	1.0	1.0	1.2	1.0	22.0
	Regular	1.0	1.0	1.0	1.4	1.0	453.0
Minor	Same-day	1.0	1.0	1.0	1.2	1.0	9.0
	Regular	1.0	1.0	1.0	1.1	1.0	80.0
Major	Same-day	1.0	1.0	1.0	1.1	1.0	8.0
	Regular	1.0	1.0	1.0	1.2	1.0	25.0

RQ3: How do client packages adopt same-day releases?

- The explicit adoption rate of same-day releases is 18% lower than that of regular releases.
- The median adoption time lag for same-day releases (9 hours) is approximately 25 times smaller than the median for regular releases (224 hours).

5.6 Discussion

In this section, we discuss the findings presented in Section 5.5. We first discuss the need for client packages to better evaluate the trade-off between the value and risk of same-day releases and how this task is currently being automated. Next, we discuss that popular package developers should plan for publishing same-day releases. We finally discuss the need for provider packages to improve the awareness of how their releases will impact client packages.

Implication 5.1) ***Third-party tools that support the automation of dependency management (e.g., Dependabot) should consider explicitly flagging same-day releases, such that client package developers become aware of the inherent trade-off between the value and risks associated with such same-day releases.*** Our preliminary study shows that same-day releases that are accompanied by release notes frequently contain important bug fixes. It is reasonable to assume that when developers fix a bug, document the fix, and publish a new release in less than 24 hours, they wish that this release will be absorbed by client packages as soon as possible. On the other hand, such a quickly implemented bug fix might have been complex, involving several code changes, thus rendering the new release particularly error prone (Nagappan and Ball, 2005). Indeed, we observed that 39% of the same-day releases have a higher code churn compared to their associated prior-to-same-day release. We also observed that 32% of the same-day releases are even followed by another same-day release. Hence, the important trade-off between the value of a same-day release (e.g., the bug fix) and its potential risks (e.g., the chances of having new bugs) should be carefully evaluated by client packages. However, in order to do so, client packages need to be able to distinguish same-day releases from regular patch releases. Given the size of the npm ecosystem and the frequent use of version range statements, it is difficult for client packages to keep track of provider updates and the introduced changes in these updates. This challenge has led to the development of tools such as Dependabot,[15] which automates dependency management. Dependabot, which has been recently acquired by GitHub, inspects the list of dependencies of a project, searches for outdated dependencies, and automatically opens individual pull requests to update each dependency (Figure 5.13). Developers of tools such as Dependabot should consider explicitly signaling same-day releases to client packages (especially if the code churn is large). This signaling would highlight the need for client packages to carefully evaluate the aforementioned trade-off between release value and risk. Dependabot, in particular, also shows the release notes associated with a dependency update. In our preliminary study, we often needed to read further documentation (e.g., issue reports) in order to fully understand the changes introduced in the same-day release. Therefore, we encourage developers of same-day releases to strive for more cohesive and clear release notes. If time pressure is too strong, developers of same-day releases can include a pointer to an external URL (e.g., GitHub or the package's website) where release notes can be carefully written after the release. The better the release notes, the easier it is for client package developers to

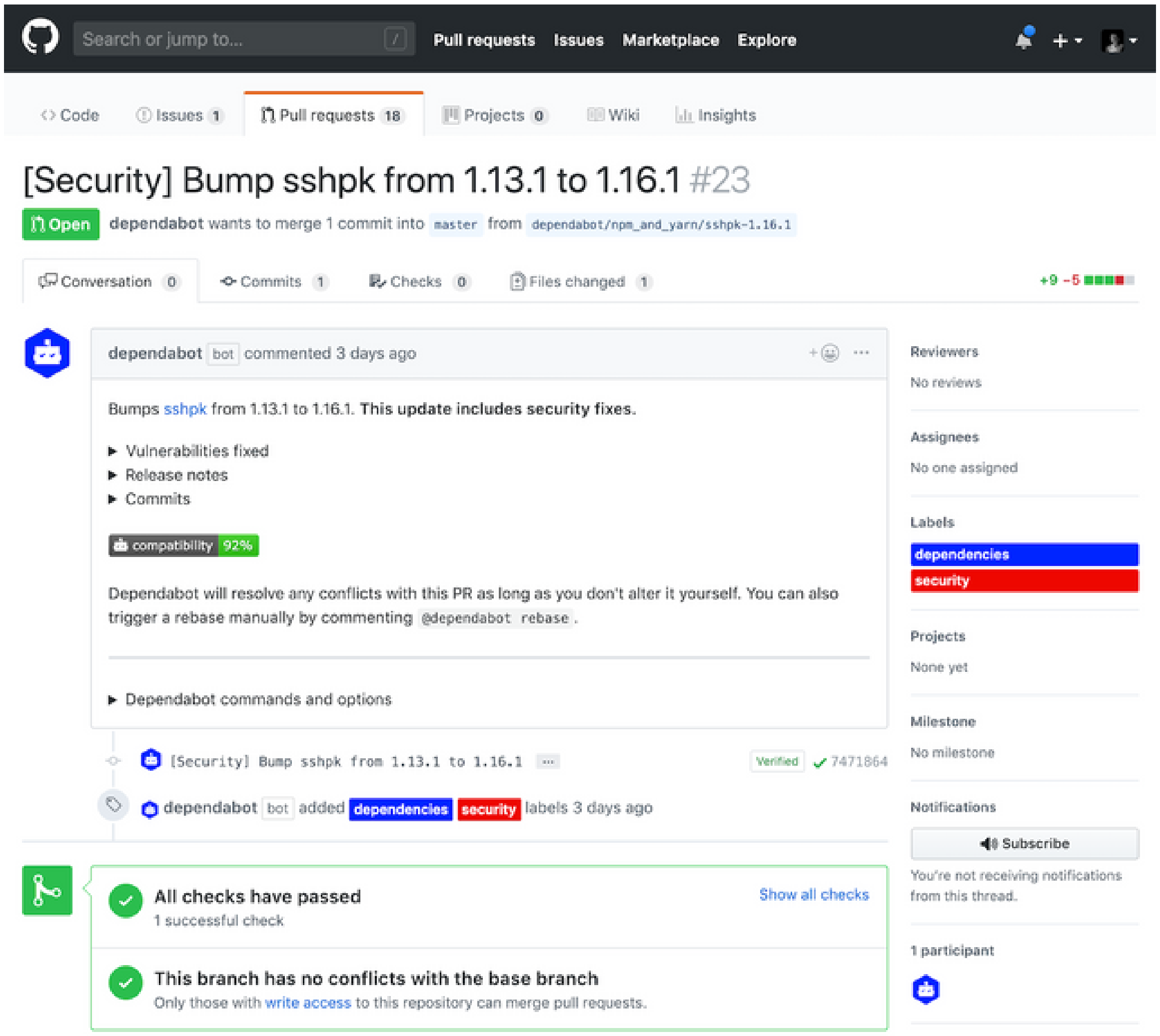
[15] https://dependabot.com

decide whether to upgrade to a same-day release or not. Finally, if a same-day release fixes a critical bug, then we recommend developers to deprecate the prior-to-same-day release in order to encourage users to perform the update.

Implication 5.2) ***Popular packages should strive for optimized release pipelines.*** In RQ1, we observed that the vast majority (96%) of the popular npm packages publish at least one same-day release and that 26% of all releases of the popular packages are same-day. Therefore, popular npm packages will benefit from being able to quickly publish releases. Although the adoption of release automation tools is a well-established software engineering practice, traditional automation tools like TravisCI[16] do not tackle ecosystem-specific problems. For instance, the constant updating of provider packages and configuration files by client packages can prevent build mechanisms to take full advantage of caching, since new provider versions or new configurations need to be reloaded. In response to this problem, npm has developed and improved upon the `npm ci` functionality[17], which aims to speed up build processes by skipping several user-oriented features that are part of the `npm install` functionality. Moreover, the `npm ci` functionality avoids updates to configuration files and individual dependencies during builds, as dependencies are deemed frozen and read from either the `package-lock.json` or `npm-shrinkwrap.json` file. Therefore, we encourage popular npm packages to have optimized release pipelines for rapid releases and to consider the use of the `npm ci` functionality.

Implication 5.3) ***Provider packages should improve the awareness of how their releases impact client packages.*** In RQ3, we observed that, although the adoption rates

[16] https://travis-ci.org
[17] https://blog.npmjs.org/post/171556855892/introducing-npm-ci-for-faster-more-reliable



[Security] Bump sshpk from 1.13.1 to 1.16.1 #23

Open dependabot wants to merge 1 commit into `master` from `dependabot/npm_and_yarn/sshpk-1.16.1`

dependabot bot commented 3 days ago

Bumps sshpk from 1.13.1 to 1.16.1. **This update includes security fixes.**

▸ Vulnerabilities fixed
▸ Release notes
▸ Commits

compatibility 92%

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

▸ Dependabot commands and options

`[Security] Bump sshpk from 1.13.1 to 1.16.1` … Verified ✓ 7471864

dependabot bot added dependencies security labels 3 days ago

All checks have passed
1 successful check

This branch has no conflicts with the base branch
Only those with write access to this repository can merge pull requests.

No reviews
No one assigned
dependencies
security
None yet
No milestone
You're not receiving notifications from this thread.
1 participant

Figure 5.13: An example of a pull request created by Dependabot. The pull request includes a list of the vulnerabilities fixed, the release notes, and commits associated with the dependency update. Image extracted from https://dependabot.com.

of same-day releases are smaller than the adoption rates of regular releases, same-day releases are adopted significantly faster. In fact, 61% of the explicit adoption of a same-day release occurs in less than 24 hours after the same-day release was published. Since same-day releases are quickly adopted by client packages (either implicitly by the usage of version range statements or even by explicit updates), it is important for popular provider packages to understand how certain code changes will impact client packages. By combining the assessment of the importance of a same-day release to client packages and how client packages are impacted by the current changes, popular provider packages can make a more informed decision about when exactly to release (or even whether field testing should be emphasized for that release). Currently, the support for assessing the impact of a provider release on its client packages is limited and the existing solutions (e.g., regularly crawling the npm registry) are not sufficient for a precise and timely assessment. Developers will, therefore, benefit from research endeavours to develop and evaluate just-in-time tools to obtain early feedback on whether (and how) their changes will propagate to client packages. Ideally, such just-in-time tools should allow provider developers to assess the impact of their changes on client packages while changes are being implemented. We also observed that updates in same-day releases often occur when both client and provider packages are maintained in the same codebase (e.g., monolithic repository), suggesting that coupling between packages can play a role in the occurrence of same-day releases. For packages maintained in the same codebase, just-in-time change propagation tools should highlight changes in the provider that drive a client package release.

5.7 Threats to validity

Construct validity. *Extraction of npm packages content:* In the preliminary study and RQ2, we analyze the files that are associated with each package release. To obtain these files, we use the `npm pack` tool, which gives us the exact files that were deployed in npm with a package release. Considering the release pipeline depicted in Figure 5.1, the files that are obtained in our data collection procedure are the files that are generated in the deployment phase. The advantage of this approach to obtain the associated files with a package release is that the file-release association is undoubtedly correct. A disadvantage is that file changes are measured in a coarse-grained, release-oriented view (German and Hindle, 2005). An alternative approach to obtaining the files associated with each package release is to collect the data from the packages' codebases (e.g., Version Control System repositories, such as Git). In this case, the obtained files correspond to the files that are generated in the revision phase. An advantage of this alternative approach is that file changes are measured in a fine-grained, commit-oriented view. In turn, as a disadvantage, the creation of parallel branches causes the commits for different releases to be chronologically interleaved (i.e., when the numerical order of a release disagrees with the chronological order). As a result, erroneously linking between an npm release and a Git commit can happen (e.g., when a timestamp-based heuristic to link npm releases and Git commits is adopted). Since our RQs are release-oriented, erroneously linking a file with a release introduces more threats to the validity of our study than assuming a coarse-grained perspective of the changes.

Identification of release notes: In our preliminary study, we identify release notes by matching file names with a regular expression. Our manual analysis allowed us to validate the correctness of this method to identify changelog files (i.e., files that contain release notes). In particular, we observed that all sampled files were correctly identified as a changelog file and are associated with the correct release. Although correct, we cannot ensure that our method has perfect recall. For instance, it is possible that certain projects use an unconventional naming scheme for their changelog files, which will not be captured by our regular expression. In addition, certain projects might decide to store release notes exclusively on external locations (e.g., a GitHub repository or project website with documentation).

We also verified whether release notes are stored in files of which the names start with the "`readme`" or "`README`" strings. We manually analyzed the content of a random sample of such files. Although we observed that a small fraction of such files do store release notes information, those files typically contain information that is related to the package's documentation (e.g., package description, installation and usage instructions), deprecation notices, and information on build status and test coverage. Therefore, we are concerned that, by considering such files as changelog files, we introduce more noise than that we enrich our data. For this reason, we do not consider the release notes that can occasionally be found in those files.

Operational definition of same-days: Packages that are published in a software ecosystem have different release schedules. Some packages publish releases on a time basis (e.g., releases are periodically published based on fixed deadlines). Other packages can publish releases on a feature basis (e.g., releases are published whenever a new feature or bug fix is implemented). For this reason, it is not trivial to define what a time-constrained release is. As an effort to provide such a definition, prior research focused on releases whose time-since-previous-release is a lower outlier in the package's time-between-releases distribution (Lin et al., 2017; Hassan et al., 2017). However, this operational definition can include long-lived releases, such as releases from packages that have a large mean time-between-releases (e.g., many months). These long-lived releases do not necessarily represent time-constrained releases. Therefore, we adopt a fixed-time approach to define time-constrained releases: releases that are published within a 24 hours time window.

In our data selection process, we select packages that have less than 50% of all releases as same-day releases. By applying such a selection criterion, our objective is to exclude from our analyses those same-day releases that are part of a regular release schedule. Same-day releases that are part of a regular release schedule resemble regular releases and, therefore, may not reflect the effects of a time restriction that we are interested in studying. Nonetheless, a different proportion other than 50% can be chosen as a threshold value to this selection criterion, depending on how conservative the definition of regular release schedule is. In specific, by assuming a 45% and a 55% threshold value as a selection criteria, we observed a difference of 100 (3%) and 216 (7%) selected packages compared with the number of selected packages using the 50% threshold value.

Operational definition of popularity: In this chapter, we focused on popular packages. Our definition of popular package is based on the number of client packages that a certain provider package has within the ecosystem. In particular, we study packages with more than 100 clients. This threshold was selected after inspecting the distribution of the total number of clients per provider package. Other measures of popularity can be considered, in particular the download count as provided by npm and the usage of npm packages by external projects to npm (e.g., open-source projects). However, the download count data that is provided by npm might not properly reflect the importance of the same-day releases of a provider package to client packages in the ecosystem (Dey and Mockus, 2018).[18] Similarly, accounting for the usage of an npm package by external projects would require a completely different research design and data collection procedures (e.g., it would require defining the external projects to be crawled, which could bias our results).

External Validity. The scope of our analysis is limited to the data from packages within the npm ecosystem (i.e., packages that are deployed in the npm registry) and thus do not include data from external applications that use an npm package. The reason for limiting our analysis to packages within the ecosystem is that new releases are directly published to and obtained from the package manager. That is, the npm registry constitutes a reliable data source that provides a ground truth instance of the released versions by each package. Although external applications also depend on npm packages, it is difficult to accurately trace the state of these external applications. In particular, a key challenge is to determine when those external applications perform a release and which version of the provider package they are using at a certain given time.

Because we collected data exclusively from npm, our findings may or may not generalize to other ecosystems. Although npm is representative in size, each software ecosystem has its intrinsic characteristics. We note that the goal of this study is not to build theories around same-day releases that would apply to all software ecosystems. Rather, our study is an important first step towards a deeper understanding of how packages in the npm ecosystem publish same-day releases. Therefore, we acknowledge that additional studies are required in order to further generalize our results.

[18] https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts

5.8 Conclusion

Same-day releases are important to maintain the ecosystem in a functional state. Although prior studies investigated the occurrence of same-day releases (Kerzazi and Adams, 2016; Lin et al., 2017; Hassan et al., 2017), this is the first study about same-day releases in a software ecosystem platform. We used data from the npm ecosystem to understand how often same-day releases are published, the changes that are performed in such releases, and how client packages react to the same-day releases of their providers. Based on our empirical observations, we provide implications that can help development teams to reason about the time constraint of same-day releases and to improve the management of development activities in such releases.

We observed that, despite the restricted time frame within which same-day releases are developed, these releases introduce important changes. Also, 96% of the popular packages published at least one same-day release. A non-negligible proportion of the same-day releases modify more lines of code compared with prior-to-same-day and regular releases. Finally, many same-day releases are usually promptly adopted by client packages (i.e., in a few hours or in the next client release).

Our findings yield three main implications. First, large changes implemented in a quick manner can render some same-day releases particularly error prone. Indeed, same-day releases are often patched: 39% of the same-day releases are followed by another same-day release. Hence, client packages need proper support to assess the trade-off between the value and the risk of adopting a same-day release. Second, the high rate of same-day releases in npm suggests that it is unavoidable to popular provider packages to eventually publish a same-day releases. Hence, popular packages will benefit from having a release pipeline that is able to process same-day releases, which might require specific configurations and tooling. Finally, since same-day releases are generally quickly adopted by client packages, provider packages will benefit to timely understand how the implemented changes impact their client packages. Our empirical observations contribute to the advance of the research concerning release management in software ecosystems. In particular, our study complements prior studies that relate same-day releases with releases that address urgent issues in the prior release.

CHAPTER 6

An Empirical Study of Deprecation of Packages and Releases

Deprecation is used by developers to discourage the usage of certain features of a software system. Prior studies have focused on the deprecation of source code features, such as API methods. With the advent of software ecosystems, package managers started to allow developers to deprecate higher-level features, such as package releases. This study examines how the deprecation mechanism offered by the npm package manager is used to deprecate releases that are published in the ecosystem. We observea that the proportion of packages that have at least one deprecated release is 3.7% and that 66% of such packages have deprecated all their releases, preventing client packages to migrate from a deprecated to a replacement release. Also, 31% of the partially deprecated packages do not have any replacement release. In addition, we investigate the content of the deprecation messages and identify five rationales behind the deprecation of releases, namely: withdrawal, supersede, defect, test, and incompatibility. We also found that, at the time of our data collection, 27% of all client packages directly adopt at least one deprecated release and that 54% of all client packages transitively adopt at least one deprecated release. The direct adoption of deprecated releases is highly skewed, with the top 40 popular deprecated releases accounting for more than half of all deprecated releases adoption.

An earlier version of this chapter is under major review in the IEEE Transactions on Software Engineering journal (TSE).

In this chapter, we describe our study about deprecation of packages and releases in npm. Section 6.1 introduces and motivates our study. Section 6.2 presents the key concepts employed throughout our study about deprecation of releases in npm. Section 6.3 explains the data collection procedures that we followed to conduct our study. Section 6.4 presents the motivation, approach, and findings from our two research questions. Section 6.5 discusses the implications of our findings. Section 6.6 presents the different perspectives from which prior research has investigated the notion of deprecation. Section 6.7 discusses threats to the validity of our study. Finally, Section 6.8 concludes the chapter by summarizing our observations.

6.1 Introduction

Deprecation is a mechanism used by developers to communicate that a software's feature is obsolete and its usage should be avoided (Zhou and Walker, 2016). Traditionally, deprecation is done at the source code level, allowing developers to deprecate any function from an API (Sawant et al., 2018a). When a function is deprecated, a compile-time warning is typically issued whenever a call to such a function is performed (Robbes et al., 2012a). In the context of software ecosystems, in which client packages depend on a specific release of a provider package, deprecation can be offered at the release level by a package manager. Therefore, developers can deprecate the entire release of a package in a software ecosystem. In such cases, an install-time warning is issued whenever a client package installs a deprecated provider release using the package manager.

Although many different aspects of API deprecation have been studied (Sawant et al., 2019; Li et al., 2018; Hora et al., 2018; Sawant et al., 2018b,a; Brito et al., 2016), the deprecation of releases in a software ecosystem was never studied. Simple characteristics such as the frequency with which releases are deprecated remain unknown. The complex network of package dependencies typically found in software ecosystems raises important concerns, such as how often deprecated releases are adopted and whether such an adoption occurs directly (by means of direct dependencies) or indirectly (by means of transitive dependencies). Client packages that directly depend on a deprecated provider release can migrate to a replacement release, i.e., update the provider to a newer version that is not deprecated. Nevertheless, this migration is not always straightforward, since replacement releases might not always exist or be easily discoverable. Indeed, the provision of a proper replacement release and its communication depend entirely on the maintainers of provider packages. Furthermore, when a deprecated provider releases is transitively adopted, the client package has no control over the migration to a replacement release. When client packages decide to continue using a deprecated release (e.g., because a migration to replacement release would incur a costly change to the codebase or lead to some incompatibility), they should be aware of the risks of doing so. For instance, a release might be marked as deprecated because it contains a defect. Yet, the rationale behind release deprecations in a software ecosystem has not been investigated by prior literature

In this chapter we study the deprecation of releases in the npm ecosystem, which is the largest software ecosystem to date.[1] In the following, we list our research questions and the key results that we obtained:

RQ1. How often are releases deprecated? Deprecation is performed by almost 4% of the packages in npm, with two-thirds of these packages being fully deprecated

[1]https://insights.stackoverflow.com/survey/2019#technology

(i.e., all their releases are deprecated). Almost one-third (31%) of the partially deprecated packages do not offer any follow-up replacement release and 15% of the existing follow-up replacement releases are major releases. Also, withdrawal (i.e., terminating the development of the deprecated package) is the most common rationale for fully deprecating a package (49%) and defect is the most common rationale for deprecating a specific release (63%).

RQ2. How do client packages adopt deprecated releases? At the time of our data collection, 27% of all client packages in npm directly adopt at least one deprecated release. Half of these adoptions target a specific set of 40 provider releases. All these 40 deprecated releases report a replacement release in their deprecation message. In addition, more than half (54%) of all client packages in the ecosystem transitively adopt at least one deprecated release. The median number of direct provider packages that result in the transitive adoption of at least one deprecated release is 1.

The main contribution of this study is to build a body of knowledge and provide insights into how releases are deprecated in a large software ecosystem, as well as into how client packages adopt such deprecated releases. Our results provide information to client packages regarding how often deprecated releases are published and the common reasons for deprecation, which help in understanding the associated risks with adopting a deprecated release. Also, client packages will find relevant information regarding the identification of transitively adopted deprecated releases and the associated challenges with migrating away from deprecated releases. Finally, we highlight the rudimentary aspect of the deprecation mechanism employed by npm and recommend a set of improvements to this mechanism. Our recommended improvements aim to support client packages in detecting and reasoning about deprecated releases. A supplementary package with our preprocessed data is available online.[2]

6.2 The deprecation mechanism of npm

Provider packages use the deprecation mechanism to maintain backward compatibility of prior releases. Instead of removing the deprecated release and causing a failure in the client packages that adopt the removed release, provider package developers opt for the deprecation. In contrast with the deprecation mechanism offered by programming languages, in which a certain method can be deprecated, the deprecation mechanism offered by a package manager allows developers to deprecate an entire release. For instance, to deprecate release 1.2.3 of a package P in npm, a developer can use the following command:[3] `npm deprecate P@1.2.3 "this release contains a bug that is fixed in version 1.2.4"`. When a release is deprecated, the npm registry is modified and the release is recorded as being deprecated (to date, the timestamp at which the deprecation is performed is not recorded in npm, making it impossible to analyze the deprecation history of a package). The deprecation mechanism adopted by npm also allows one to deprecate a range of versions or, alternatively, the entire package (which essentially deprecates all versions of the package). In such cases, the installation of any version that satisfies the deprecated version range will issue a deprecation warning.

Information about the deprecation of a release can be stated in the *deprecation message*. In the prior example, the deprecation message states that the reason for the

[2]https://bit.ly/2wKO3se.
[3]https://docs.npmjs.com/cli/deprecate

deprecation is the presence of a bug in release 1.2.3. Whenever a deprecated release is installed by some user, a warning is issued (at the installation time) and the deprecation message is displayed. For example, when the command `npm install P@1.2.3`[4] is used to install release 1.2.3 of package P, the installation will succeed and the deprecation message will be displayed. Whenever a client package that adopt a deprecated provider release is installed, a warning is issued and the provider's deprecation message is displayed. For example, suppose that a client package C adopts the deprecated release 1.2.3 of provider P. Whenever the command `npm install C@latest` is used to install the latest release of package C, a warning with the deprecation message of release 1.2.3 of package P will be displayed.

Any deprecated provider release that is adopted by means of a transitive dependency also yields a warning. Deprecation warnings that come from a transitive dependency are difficult to trace,[5] since the issued warning does not explicitly indicate the dependency depth. By dependency depth, we mean the number of downstream dependencies from one client package to a transitively adopted provider. For example, if a client package C depends on a provider P_1 that, in turn, depends on a provider P_2, then the transitive dependency between C and P_2 has a depth of 2.

Client packages might want to migrate away from a deprecated provider release towards some *replacement release* (i.e., a non-deprecated provider release that follows the deprecated one). Client packages that set a dependency using a version range statement will eventually perform an implicit update to the replacement release, as long as the version range is satisfied by the replacement release. Nevertheless, in many

[4] https://docs.npmjs.com/cli/install

[5] https://stackoverflow.com/questions/36329944/how-to-determine-path-to-deep-outdated-deprecated-packages-npm

cases, the migration to a replacement release requires modifying the versioning statement. Because the installation of a client package yields the installation of all transitive dependencies, updating a deprecated provider package that is transitively adopted can also be desired.[6] However, identifying whether the transitively adopted deprecated release has a follow-up replacement release (and should consequently be updated) might not be trivial. In particular, npm provides the `npm outdated`[7] tool that checks which providers can be updated. To check transitive dependencies, this tool requires an argument that determines the maximum dependencies depth to be checked. For example, `npm outdated --depth 1` will check whether the providers of the direct providers can be updated. To date, the value for this argument needs to be determined by a trial-and-error approach, i.e., iteratively increasing the value of the `--depth` argument until all transitive dependencies are checked.

In Table 6.1, we summarize the deprecation-related terms that are used throughout this thesis.

6.3 Data collection

In this section, we describe how we collected data from npm to our study about releases deprecation. Figure 6.1 depicts an overview of our data collection method. Three main steps are performed: collect package metadata (Section 6.3.1), analyze package releases (Section 6.3.2), and analyze package dependencies (Section 6.3.3). In the following, we discuss each of these steps.

[6] https://stackoverflow.com/questions/35236735/npm-warn-message-about-deprecated-package/36341065

[7] https://docs.npmjs.com/cli/outdated.html

Table 6.1: Summary of the deprecation terminology used in this chapter.

Term	**Definition**
Deprecated release	A package release that is deprecated at the time of our data collection.
Non-deprecated release	A package release that is not deprecated at the time of our data collection.
Deprecation message	A message that is warned to client packages when they install a deprecated release. The message is provided by the developer of the deprecated package
Newest deprecated release	The deprecated package release with the largest version number within the sequence of package releases.
Older non-deprecated release	The non-deprecated replacement release with the largest version number that is smaller than the newest deprecated release version number.
Fully deprecated package	A package with all of its releases deemed as deprecated.
Partially deprecated package	A package that has some of its releases as deprecated, but not all releases.
Replacement release	A non-deprecated release that can replace a deprecated release. For partially deprecated packages, a replacement release can be automatically identified as the release whose version number is the next after the newest deprecated release.
Replacement package	A package deemed as a replacement for a fully deprecated package.

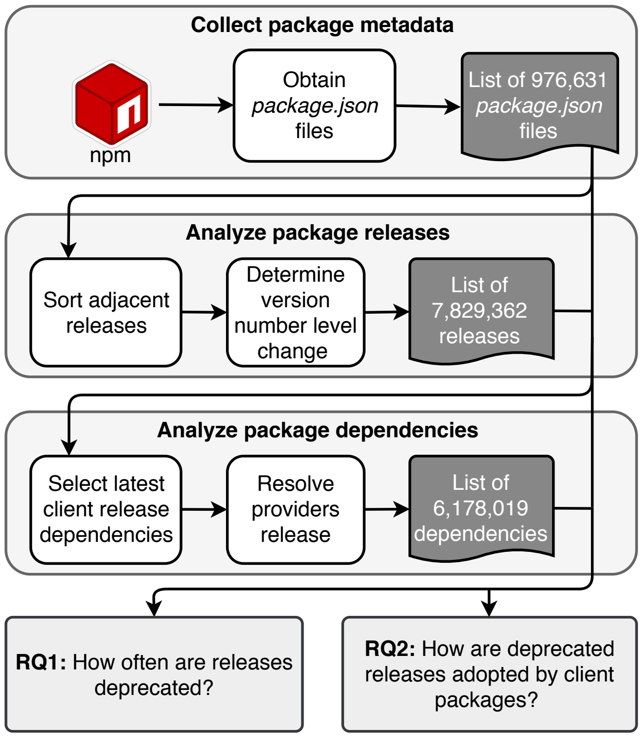

Collect package metadata
npm
Obtain *package.json* files
List of 976,631 *package.json* files
Analyze package releases
Sort adjacent releases
Determine version number level change
List of 7,829,362 releases
Analyze package dependencies
Select latest client release dependencies
Resolve providers release
List of 6,178,019 dependencies
RQ1: How often are releases deprecated?
RQ2: How are deprecated releases adopted by client packages?

Figure 6.1: An overview of our data collection method.

6.3.1 Collect package metadata

In this step, we collect the metadata of npm packages from the `package.json` files (see Chapter 2 for an explanation of the `package.json file`).

Obtain `package.json` files: We obtained all `package.json` files that were stored in the npm registry as of May, 2019. The `package.json` file of a given package P contains release-related metadata for all releases of P. The release-related metadata include the release version number, the timestamp at which the release was published, the dependencies that are set in the release (i.e., the provider package name and the respective versioning statement), and the deprecation message in case the release is deprecated. The output of this procedure is a list of 976,631 `package.json` files.

When the release of a package P is deprecated, the `package.json` file of P contains a `deprecated` field that stores the deprecation message that is associated to that release (in contrast, non-deprecated releases do not have such a field). We observed that 8% of all deprecated releases have the "False" string as a deprecation message. After performing a manual analysis (see Section 6.4.2 for further details about this analysis), we decided to classify such releases as non-deprecated. Also, 1% of the deprecated releases have an empty deprecation message. According to the npm documentation on deprecation,[8] a developer can remove the deprecation of a release by setting the deprecation message to an empty string. Therefore, releases whose deprecation message is an empty string were classified as non-deprecated.

[8]`https://docs.npmjs.com/deprecating-and-undeprecating-packages-or-package-versions`

6.3.2 Analyze package releases

In this step, we sort the adjacent package releases and determine how the version numbers change between two adjacent releases (e.g., whether the latter release is a major, minor, or patch release). See Chapter 2 for a definition of the release version levels.

Sort adjacent releases: We sort the releases of a package according to a *branch-based ordering* (in contrast with a chronological- or numerical-based ordering) as described in Section 4.2. The motivation for using a branch-based ordering is the adoption of parallel development branches by some packages, for which chronologically interleaved releases are published (e.g., release 1.2.3, a patch for release 1.2.2, being published after the existence of release 2.0.0). With a chronological-based ordering, release 2.0.0 would be considered the predecessor of release 1.2.3, which is incongruous from the numerical point of view. Similarly, with a numerical-based ordering, release 1.2.3 would be considered the predecessor of release 2.0.0, which is incongruous from the chronological point of view (after all, release 1.2.3 was published *after* release 2.0.0). A branch-based ordering schema allows a release to be considered the predecessor of more than one release. In our example, release 1.2.2 would be considered the predecessor of both releases 1.2.3 and 2.0.0.

Determine version level change: After sorting the releases of a package according to our branch-based ordering, we analyze how the version level changes between two adjacent releases. For each pair of adjacent releases, we classify the version level change into a major, minor, or patch release. For example, if release 1.2.2 is considered the predecessor of release 2.0.0, then release 2.0.0 is classified as a major release. Similarly, if release 1.2.2 is considered the predecessor of release 1.2.3, then release 1.2.3 is

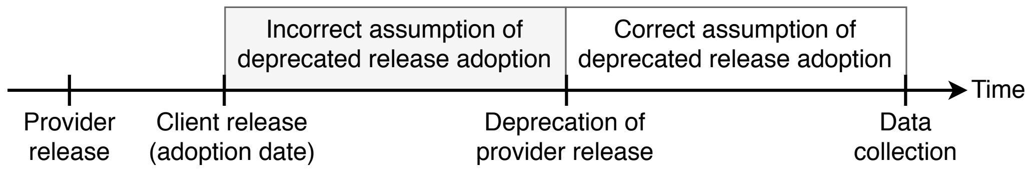

Incorrect assumption of deprecated release adoption
Correct assumption of deprecated release adoption
Time
Provider release
Client release (adoption date)
Deprecation of provider release
Data collection

Figure 6.2: Illustration of a scenario in which the provider release is deprecated after it is adopted by the client package.

classified as a patch release. The same logic applies to minor releases. The output of this procedure is a list of 7,829,364 release changes.

6.3.3 Analyze package dependencies

In this step, we select a subset of all dependencies, namely the dependencies that are set in the latest client release. We also resolve the release of the providers that are used by each client package (see Chapter 2 for a description of how a provider version is resolved).

Select latest client release dependencies: The date at which a provider package release was deprecated is not available in the npm registry. This limitation in the npm data prevents an analysis about the adoption history of deprecated releases by client packages. For example, Figure 6.2 depicts a scenario in which the provider release is deprecated after it is adopted by a client package. In such case, we can only correctly assume that the client package is using a deprecated provider release at our data collection date (since the deprecation date is unknown). Therefore, we select only the latest client release (i.e., the current client release at our data collection date) to analyze how deprecated provider releases are adopted by client packages.

Resolve providers release: We resolve the release of the providers that are used in the latest release of each client package. For the latest release of each client package, we parse the used versioning statements according to the grammar provided by `npm` (see Chapter 2 for a description of this grammar). The resolved release is the latest provider release (at the time of the client release) that is satisfied by the versioning statement. For instance, suppose that a client package C, in its 2.0.0 release, sets a dependency using the versioning statement `"P: >1.2.3"`. In this case, the resolved provider release will be the latest release of P (the provider package) that is greater or equal 1.2.3 and that is published before the client package release 2.0.0. Invalid versioning statements or versioning statements that do not satisfy any existing provider release are ignored. The output of this procedure is a list of 6,178,019 dependencies that are set in the latest client release with the respective resolved release of each provider.

6.4 Results

In this section, we present the results of our two RQs. For each RQ, we discuss its motivation, the approach that we used to address it, and the results that we obtained.

6.4.1 RQ1: How often are releases deprecated?

In this RQ, we investigate the frequency with which releases are deprecated. We also investigate the reasons behind the deprecation of a package or a release and estimate how often each reason is stated in a deprecation message.

6.4.1.1 Deprecation frequency

Below, we describe the motivation, approach, and results for our study about deprecation frequency.

Motivation. Deprecation can occur at different levels of granularity within a software system. Traditionally, prior studies focus on API deprecation, which typically operate at the function level (Li et al., 2018; Hora et al., 2018; Sawant et al., 2018c; Zhou and Walker, 2016; Robbes et al., 2012a). However, the offered deprecation mechanism by the npm package manager operates at a higher level, allowing one to deprecate either a release, a range of releases, or the entire package (i.e., all releases of the package). To the best of our knowledge, no prior studies have investigated how often deprecation occurs at the release-level within a software ecosystem. Therefore, client packages will benefit from understanding the reasons behind deprecation, as well as the frequency with which releases are deprecated for each of such reasons.

Ideally, the deprecation of a release should occur after a replacement release is published (i.e., some newer release that is not deprecated). In such cases, client packages that are using a deprecated provider release can perform an update targeting the replacement release. Therefore, it is important for client packages to know how often follow-up replacement releases are available. Also, client packages would benefit from having an estimate of the needed effort to perform an update to a replacement release (i.e., knowing whether replacement releases are patch, minor, or major releases). Nonetheless, when a replacement release is not available, client packages that want to migrate away from a deprecated release need to perform a downgrade (i.e., migrate to an older release that is not deprecated). The drawback of performing a downgrade is that client packages can miss new features and bug fixes from newer releases (Cogo et al., 2019; Decan et al., 2018). Hence, it is interesting for client packages to understand how many patch, minor, and major releases are back skipped when a downgrade from a deprecated release to an older non-deprecated release is performed. For example, a downgrade that changes the resolved provider release from 1.0.1 to 1.0.0 is back skipping one patch release.

Approach. To determine the prevalence of release deprecation in npm, we calculate the proportion of packages that have at least one deprecated release and the proportion of all releases that are deprecated in the ecosystem. We also determine the prevalence of packages that are *fully deprecated* (i.e., all releases are marked as deprecated) or *partially deprecated* (i.e., some releases are marked as deprecated, but not all). To this end, for each package, we calculate the proportion of deprecated releases over all its published releases.

Next, we study how often deprecated releases have a follow-up *replacement release*. More specifically, we calculate the proportion of partially deprecated packages that have some non-deprecated release that follows the *latest deprecated release* (i.e., the deprecated release with the largest version number within a package's sequence of releases). We focus on the latest deprecated release because we want to understand the availability of replacement releases as of the data collection date (since we do not have historical deprecation information). As an example, Figure 6.3 shows a hypothetical sequence of releases of a partially deprecated package with a replacement release. The 2.1.1 release is the latest deprecated release and the following 3.0.0 release is the replacement release. Furthermore, when a replacement release is not made available by a partially deprecated provider package (i.e., the provider package has deprecated its latest release), client packages can downgrade the provider package by adopting an *older non-deprecated release* to replace the deprecated release. In our approach, the older non-deprecated release is the largest non-deprecated release whose version number is smaller than the latest deprecated release (see Figure 6.3).

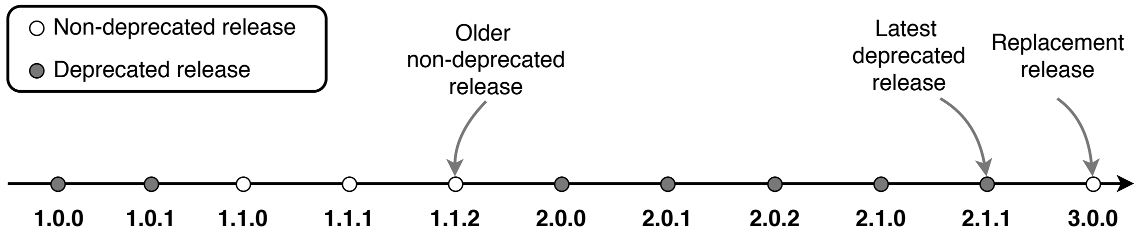


Figure 6.3: Older non-deprecated and replacement release of a hypothetical partially deprecated package.

We estimate the technical lag (Zerouali et al., 2019b) that is induced when client packages downgrade from the latest deprecated provider release to the older non-deprecated release (e.g., downgrading the adopted provider release from 2.1.1 to 1.1.2 in the example of Figure 6.3). Such an estimation is done by counting the number of major, minor, or patch releases that are back skipped when the downgrade occurs. More specifically, we determine the highest version level among the releases that are published in between the older non-deprecated release and the latest deprecated release (i.e., either major, minor, patch levels) and then we count how many releases of such level were published during this time frame. For example, in Figure 6.3, in between the older non-deprecated release and the latest deprecated release, one major release is published (2.0.0), one minor release is published (2.1.0), and three patch releases are published (2.0.1, 2.0.2, 2.1.1). In this scenario, if a client package that adopts the latest deprecated release decides to downgrade to the older non-deprecated release, this client will back skip one major provider release.

Finally, we calculate the proportion of replacement releases that are major, minor, or patch releases (in the example shown in Figure 6.3, the replacement release is a major release).

Observation 6.1) ***3.7% of the npm packages have at least one deprecated release.*** There are 253,501 deprecated releases in npm (3.2% of all releases) and 31,810 packages with at least one deprecated release in npm, representing 3.7% of all packages in the ecosystem. Two-thirds (66%) of the packages with a deprecated release are fully deprecated (i.e., have deprecated all releases). Figure 6.4 shows the proportion and the total number of deprecated releases per package with at least one deprecated release. A total of 29% of the fully deprecated packages have a single release (represented by the darkest portion of the largest bar in Figure 6.4) and 20% have ten or more deprecated releases (represented by the sum of the two lightest portions of the largest bar in Figure 6.4). Among the partially deprecated packages, 69% have more than one deprecated release. That is, partially deprecated packages tend to either deprecate a range of releases or apply the deprecation mechanism more than once over time. We cannot distinguish between these two cases because the `package.json` files do not record historical information about deprecation (i.e., one can only know whether a certain release is deprecated or not).

Observation 6.2) ***31% of the partially deprecated packages do not have a replacement release.*** When a follow-up replacement release is not available, client packages can only migrate away from deprecated releases by downgrading the adopted provider. Figure 6.5 shows the number of major, minor, and patch releases between the older non-deprecated release and the latest deprecated release. In total, 20% of the partially deprecated packages published one or more major release between the older non-deprecated and the latest deprecated release. In turn, 40% of such packages published one or more minors (but no majors) between the older non-deprecated and the latest deprecated release. Finally, 40% of these packages have one or more patches

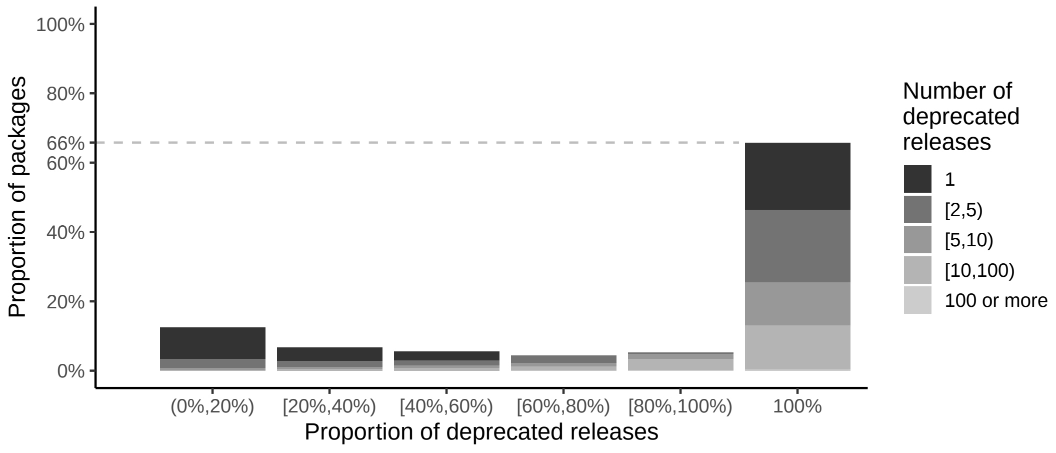


Figure 6.4: Number and proportion of deprecated releases per package (excluding packages without deprecated releases). The dashed line shows the proportion of fully deprecated packages.

(but no majors or minors) between the older non-deprecated and the latest deprecated release. As shown in Figure 6.5, the median is one for the three release levels.

Observation 6.3) ***68% of the replacement releases are patch releases, 17% are minors, and 15% are majors.*** Although the majority of the replacement releases introduce simpler changes (patch and minor releases), a non-negligible number of the replacement releases introduce more complex changes (i.e., major releases, which might introduce backward incompatible changes). Client packages must be aware of the possibility of having to integrate major releases of the providers when they are willing to adopt a replacement release. Our analysis shows that even client packages that set a restrictive range for their versioning statements (e.g., version ranges that accept only patch updates) will likely perform an implicit update to the replacement release.

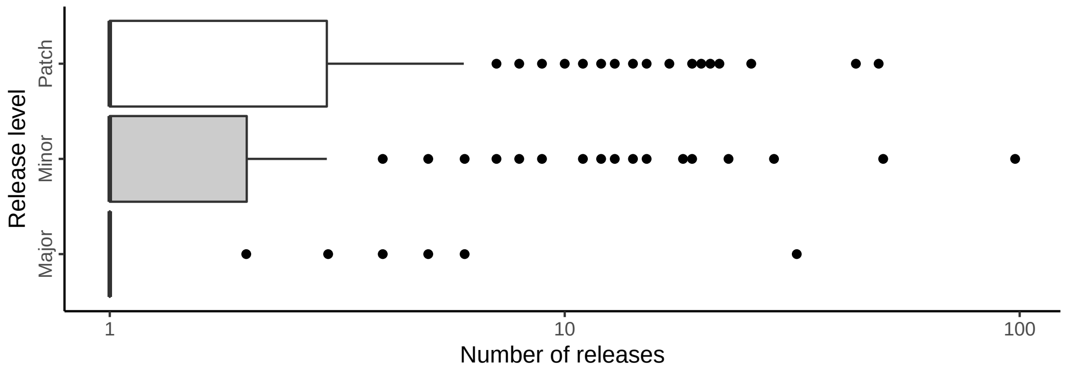


Figure 6.5: Number of major, minor, and patch releases that are published between the older non-deprecated release and the latest deprecated release of packages without a replacement release.

6.4.1.2 Rationales for deprecation

Below, we describe the motivation, approach, and results for our study about the rationales behind deprecation of packages and releases.

Motivation. Documentation is an important aspect of deprecation. With proper deprecation messages, client packages can understand the reason for the deprecation and evaluate the risk of adopting a given deprecated release. Furthermore, it is important that deprecation messages report the replacement packages and releases, therefore client packages can perform an easier migration.

Approach. We manually analyzed a statistically representative sample (95% confidence level, with ±5% confidence interval) of the deprecation messages used by npm packages. We sampled a total of 381 out of the 44,112 unique deprecation messages used by different packages. The performed analysis resulted in a categorization of the rationale behind the deprecation of a release, as well as an estimate of the proportion of deprecation messages that report a replacement package or release. We performed an open coding (Stol et al., 2016) to categorize the rationale behind release-level deprecation. We also classify the deprecation messages between messages that refer to a deprecated package (e.g., "*This package is no longer supported*") from messages that refer to a specific release of the deprecated package (e.g., "*This version has a bug. Use version 1.0.1 instead*"). We perform such a classification to calculate how often each identified rationale is associated with the deprecation of all releases of a package (full deprecation), in contrast to the deprecation of a specific release of a package (partial deprecation). Finally, we calculate how often a deprecation message reports a replacement package or a replacement release.

Observation 6.4) ***Almost two-thirds (64%) of the deprecation messages report the rationale behind the deprecation of a package or release.*** This observation shows that, when installing a deprecated package or release, client packages in many cases will be able to evaluate the risk of adopting a deprecated release. From the deprecation messages that report the rationale behind the deprecation, 86% are a customized message (i.e., they are different from the standard message that is provided by npm).[9] This latter observation suggests that the typical rationale for the deprecation of a package or release goes far beyond the rationale stated in the standard message (which is "*Package no longer supported. Contact support@npmjs.com for more info.*" at the time this thesis was written).

We also note that a total of 8% of all deprecation messages are the string "False". To understand the usage of the "False" string as a deprecation message, we manually analyzed the revisions (i.e., the history of versions) of the `package.json` files from packages with such deprecation message. We identified that some packages have the

[9] https://docs.npmjs.com/deprecating-and-undeprecating-packages-or-package-versions

"False" deprecation messages since its creation, having never been in fact deprecated. We hypothesize that some developers might not understand the meaning of the `deprecated` field in the `package.json` file. In such cases, the developer manually edit the `package.json` file and set the `deprecated` field as "False", with the intent of communicating that the release is not deprecated. This observation suggests that the deprecation mechanism is not intuitive, leading some package developers to misunderstand how the mechanism works.

Observation 6.5) ***Withdrawal is the most common rationale for the deprecation of a package (49%) and defect is the most common rationale for the deprecation of a release (63%).*** Five rationales behind the deprecation of a package or release can be identified: withdrawal (e.g., the development of a package is no longer maintained in npm), supersede (e.g., a deprecated release is replaced by a newer, improved release), defect (e.g., a certain functionality is discovered to be buggy), test (e.g., package is published for test purposes only), and incompatibility (e.g., dependency incompatibility). The proportion with which a given rationale is associated with the deprecation of either a package or release is shown in Table 6.2. In total, 80% of the deprecation messages are for deprecated packages, whereas 20% of the deprecation messages are for deprecated releases. In the deprecation messages for deprecated packages, 51% report a replacement package. Similarly, in the deprecation messages for replacement releases, 51% report a replacement release.

Below, we describe in details each of the identified rationales behind deprecation.

Rationale 1) ***Withdrawal:*** The deprecation message indicates that the package or release was deprecated because its development was terminated. However, the package is left on the registry, such that the actual client packages are not affected. An analysis

Table 6.2: The proportion with which each rationale is associated with the deprecation of a package or a release.

Rationale for deprecation	**Package deprecation (80%)**	**Release deprecation (20%)**
Withdrawal	49.0%	12.0%
Supersede	45.0%	20.0%
Defect	0.5%	63.0%
Test	5.0%	2.5%
Incompatibility	0.5%	2.5%

of such deprecation messages shows that withdrawals occur for different reasons. The following deprecation messages indicate that the withdrawal might occur because the package/release is no longer maintained in npm:

> "*This module is no longer maintained.*" [deprecation message of package `kilt`],
>
> "*This project is no longer a npm-package. Checkt [sic] our github at https://github.com/Server-Eye/bucket-collector*" [deprecation message of package `bucket-collector`],
>
> "*Package unsupported. Please use the rws-compile-sass package instead.*" [deprecation message of package `custom-rws-compile-sass`]

The following deprecation messages indicate that the withdrawal might occur because the package/release is a dependency that is no longer required (e.g., its features were incorporated into another package/release):

"*Deprecated as it's now the default reporter in ESLint.*" [deprecation message of package `eslint-stylish`],

"*This is a stub types definition. p-limit provides its own type definitions, so you do not need this installed.*" [deprecation message of package `@types/p-limit`],

"*No longer needed for grunt-vows-runner. Use grunt-vows-runner instead.*" [deprecation message of package `vows-reporters`]

Also, as shown in the following deprecation messages, the withdrawal might occur because the package/release became obsolete.

"*Very old and unmaintained module. Don't recommend using this anymore.*" [deprecation message of package `grunt-copy-mate`],

"*Since Catberry@4 this package is not supported due to architecture changes.*" [deprecation message of package `catberry-lazy-loader`],

"*Do not use this package to update globally installed CLIs anymore.*" [deprecation message of package `npm-update-module`]

Rationale 2: ***Supersede:*** The deprecation message indicates that the deprecated package or release was replaced by another one. The following deprecation messages indicate that the deprecated release was replaced by a newer, improved release:

"*Version 1.x branch of Iridium has been superseded by v2.x.*" [deprecation message of package `iridium`],

"*Still using old declarative binding syntax? Please, update to its latest version: 0.5.102.*" [deprecation message of package `pacem`],

"*API changed: then() to on(), catch() to onerror(), finally() to oncancel().*" [deprecation message of package `rnr`]

In addition, the following deprecation messages indicate that the deprecated package features were incorporated into another package:

> "*This has been merged back into express-batch, which you should now use.*" [deprecation message of package `express-batch-deep`],
>
> "*This module is now a part of babel-preset-steelbrain@2.x.x.*" [deprecation message of package `babel-preset-steelbrain-async`],
>
> "*@appnest/focus-trap has moved to @a11y/focus-trap. Please uninstall this package and install @a11y/focus-trap instead.*" [deprecation message of package `@appnest/focus-trap`],
>
> "*All Pivotal UI components & styles have been moved to the 'pivotal-ui' package. Install that package for all future updates.*" [deprecation message of package `pui-react-checkbox`],
>
> "*This library has been renamed to flum. Please install flum to get the lastest [sic] version.*" [deprecation message of package `react-basic-forms`]

Rationale 3: ***Defect:*** The deprecation message indicates that the package or release was deprecated due to the presence of a known defect. An analysis of such deprecation messages shows that the source of defect can be either in the source code or in the deployed artifact (built package). The following deprecation messages indicate that the package or release was deprecated due to a defect in the source code:

> "*This patch version has breaking changes. Please use 0.23.0 instead.*" [deprecation message of package `@devexperts/react-kit`],
>
> "*Buggy implementation of class mixins.*" [deprecation message of package `@zenparsing/skert`],
>
> "*windows posix socket bug*" [deprecation message of package `node-ipc`],
>
> "*Sending Blob body using XMLHttpRequest polyfill may cause incorrect result with this version, please use 0.9.1 instead.*" [deprecation message of package `react-native-fetch-blob`]

Also, the following deprecation messages indicate that the package or release was deprecated due to a defect in the built package:

> "*wrong build.*" [deprecation message of package `dc-webapi`],
>
> "*critical dir missing.*" [deprecation message of package `angular-html5`],
>
> "*incorrect main field in package.json, fixed in 1.0.1*" [deprecation message of package `eslint-config-r29`],
>
> "*error in version number.*" [deprecation message of package `react-native-aerogear-ups`],
>
> "*Main script path incorrect. Only ES6 module is working.*" [deprecation message of package `defy`]

Rationale 4: *Test:* The deprecation message indicates that the deprecation occurs because the package or release was published for test purposes or by accident:

> "*This is package is just for testing. don't install it.*" [deprecation message of package `reactmanishbot`],
>
> "*Not a usable package.*" [deprecation message of package `glarce-combo`],
>
> "*not meant to be published sorry.*" [deprecation message of package `chat-engine`]

The following deprecation messages indicate that pre-releases, which are used for in-field testing, are also deprecated:

> "*Development versions have been deprecated.*" [deprecation message of package `@servicensw/page`],
>
> "*outdated prerelease.*" [deprecation message of package `@dandi/common`],
>
> "*Use version 1.0.0, this was a prerelease and is no longer maintained.*" [deprecation message of package `vue-cli-plugin-git-describe`],
>
> "*still in beta.*" [deprecation message of package `chat-engine`]

Rationale 5: ***Incompatibility:*** The deprecation message indicates that the package or release was deprecated due to incompatibility. The following deprecation messages indicate incompatibility between client and provider packages (dependency incompatibility):

> "*Old versions not compatible with sqb >0.7.0.*" [deprecation message of package `sqb-serializer-oracle`]

Also, the following deprecation messages indicate incompatibility between the package and some specific browser version:

> "*Incompatible with modern browsers.*" [deprecation message of package `yahoo-shapes`]

RQ1: How often are releases deprecated?

- More than 3% of all studied releases are deprecated, with almost 4% of all packages having at least one deprecated release.
- 66% of the packages with a deprecated release are fully deprecated.
- 31% of the partially deprecated packages do not have a replacement release.
- The most common reason to fully deprecated a package is abandoning its development (49%), whereas the most common reason to deprecate a specific release is a bug in this release (63%).

6.4.2 RQ2: How do client packages adopt deprecated releases?

In this RQ, we investigate how client packages adopt deprecated releases. We differentiate between direct and transitive adoptions (see Chapter 2 for a definition of direct and transitive dependencies). Direct adoptions are under the control of the client package, since they originate from direct dependencies (i.e., those specified in the `package.json` file). For these direct adoptions, we determine the frequency with which they happen and how they relate to the type of versioning statements that are employed by the client packages. Transitive adoptions of deprecated releases happen indirectly and thus are not under the control of client packages. As part of this RQ, we also determine the frequency with which clients transitively adopt deprecated releases, as well as how deep these adoptions happen in the dependency tree. Direct adoptions are discussed in Section 6.4.2.1 and transitive adoptions are discussed in Section 6.4.2.2.

6.4.2.1 Direct adoption of deprecated releases

Motivation. Deprecation mechanisms are employed to discourage the usage of a certain piece of code. Investigating how frequently client packages directly adopt deprecated releases will provide insights into how effective the deprecation mechanism is in avoiding client packages from adopting deprecated provider releases. In particular, it is important to understand whether there are deprecated releases that are still adopted by a large number of clients and whether a replacement release (or replacement package) exists for them. The latter is particularly relevant for client packages that value keeping their providers up-to-date.

Approach. We calculate the proportion of client packages that directly adopt at least one deprecated provider release. We then investigate whether there are deprecated releases that are adopted by more client packages than other deprecated releases. To this end, for each deprecated release d, we calculate a_d, which is the number of times that d is directly adopted by a client package. We then divide a_d by the total number of direct adoptions of deprecated releases, obtaining the proportion p_d of direct adoptions of the deprecated release d. We define a *popular deprecated release* as any release d that belongs to the smallest subset of deprecated releases for which p_d sums up to 50% (i.e., the deprecated releases that concentrate half of all adoptions). The adoption of deprecated provider releases was assessed only at the latest client release and, as a consequence, our observations are based in adoptions that occur at the time of our data collection (see Section 6.3). For the sake of simplicity, we will refer to a “client package release” simply as a “client package” (implicitly referring to the latest client package release).

Next, we determine how often popular deprecated releases have a replacement release. Because a popular deprecated release can be a release of either a partially or a fully deprecated package (see Section 6.4.1), we employ a different method for those two cases. Basically, for popular deprecated releases of fully deprecated packages, we search for a *replacement package* instead of a replacement release. In the following, we describe the two employed methods:

- *Partially deprecated packages:* To determine whether a popular deprecated release of a partially deprecated package has a replacement release, we search for the existence of any non-deprecated release whose version number is larger than the newest deprecated release (see Figure 6.3).

- *Fully deprecated packages:* To determine whether a popular deprecated release of a fully deprecated package has a replacement package, we perform a manual analysis over the deprecation messages. We search for replacement packages by reading the deprecation message of the popular deprecated releases and any documentation that is mentioned in such deprecation messages (e.g., tutorials that explain how client packages should perform changes to migrate away from the deprecated release).

After determining the replacement releases and packages, we estimate the date at which a popular deprecated release was deprecated. Such an estimate is performed to safeguard the validity of our analysis, since a given deprecated release might be massively used because client packages did not have enough time to migrate away from this release (e.g., for releases that were deprecated at a date that is close to our data collection). We estimate the deprecation date by gathering two pieces of evidence. First, we verify whether any mentioned documentation in the deprecation message includes the date at which the release was deprecated or some event that drove the deprecation and whose date can be obtained (e.g., the first stable release of a reported replacement package). Second, when none of these pieces of information are reported in the existing documentation, we assume that the deprecation occurred at the package's latest release date.

Finally, to understand how the versioning statements used by client packages relate to the adoption of a deprecated release, we analyze how client packages of the popular deprecated releases set their versioning statements. We calculate the proportion of specific version and version range statements that are used by the client packages. For version range statements, we calculate how often the tilde, caret, or the latest operator are used (see Chapter 2 for a description of the operators).

Observation 6.6) ***27% of the client packages directly adopt at least one deprecated provider release.*** In RQ1, we observed that only a small proportion of npm packages (3.7%) deprecated at least one release. Yet, when analyzing client adoption of deprecated releases, we note that 27% of all client packages in npm adopt at least one deprecated release.

Observation 6.7) ***A remarkably small proportion of the deprecated releases are massively adopted by client packages.*** More specifically, 75% of all adoptions of deprecated releases concentrate on only 2.6% of all deprecated releases (Figure 6.6). The top 40 most frequently adopted deprecated releases account for 50% of all adoptions of deprecated releases. We call these 40 deprecated releases as popular deprecated releases. In total, 80% (32 out of 40) of the popular deprecated releases are from a fully deprecated package and 20% (8 out of 40) are from a partially deprecated package.

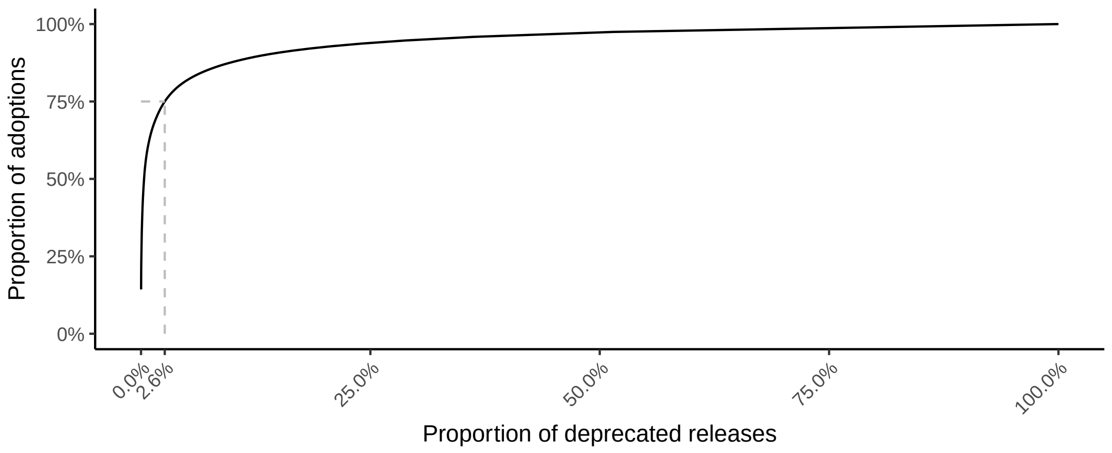


Figure 6.6: Cumulative histogram for the proportion of client packages that depend on a deprecated provider release.

Observation 6.8) ***All popular deprecated releases have a deprecation message that indicates a replacement package or release.*** This observation indicates that popular provider package developers always support clients in determining candidate releases (or packages) to be migrated to in face of release deprecation. Furthermore, we estimate that all popular deprecated releases have been marked as such for at least 6 months since our data collection date, showing that client packages had a reasonable amount of time to migrate away from them. In Table B.2 (Appendix B), we list all the 40 popular deprecated releases, their respective replacement release or package, and our estimate of their deprecation date.

Observation 6.9) ***Client packages set a version range statement in 91% of the adoptions of a popular deprecated release.*** However, the majority (80%) of the popular deprecated releases provide a replacement package (not a replacement release), which suggests that many client packages do not want to pay the cost to integrate a replacement package (i.e., replacing the deprecated provider by a whole different provider package). For these cases, the versioning statement that is used by the client package

Table 6.3: The proportion of each versioning statement type used in the adoption of the popular deprecated releases.

Versioning statement type		**Proportion**
Range	Caret (∧)	84.0%
	Tilde (∼)	3.5%
	Latest (`*, latest, >, or >=`)	2.5%
	Others	1.0%
Specific		9.0%

does not make any difference regarding the adoption of a replacement package. Table 6.3 shows the proportion of each used version range operator in the version range statements when a popular deprecated release is adopted. The large majority of the versioning statements (84%) use a caret operator, implicitly updating any patch release that is equal to or larger than 0.1.0 or any minor release that is equal to or larger than 1.0.0. Hence, in many cases, the usage of version range statements that satisfy minor and patch provider releases is not sufficient to implicitly migrate to a replacement release. In other words, unless clients modify their versioning statement, they will not migrate away from the deprecated release. Indeed, only 2.5% of the versioning statements implicitly accept major updates of the provider.

6.4.2.2 Transitive adoption of deprecated releases

Motivation. While client packages can choose the provider packages that are directly adopted, provider packages that are transitively adopted are out of the scope of client packages. As a consequence, tracking the transitive adoption of a deprecated release can be challenging to client packages. Even when providers that are transitively adopted can be tracked, the provided tools by npm that help client packages to check whether such providers can be updated require the specification of the dependency depth parameter (see Section 6.2), which is usually unknown. Also, there is no trivial approach to update transitive providers. These issues show how challenging dealing with the transitive adoption of deprecated provider releases is. By knowing how often deprecated provider releases are transitively adopted, client packages can estimate the likelihood of having to deal with a transitively adopted deprecated release. Also, transitive providers can be updated as a consequence of the update of a direct provider. Hence, client packages can benefit from estimating how often a directly adopted provider package results in the transitive adoption of a deprecated release. In this RQ, we determine how often client packages transitively adopt a deprecated provider release, what the typical dependency depth is, and how often a provider that is directly adopted results in the transitive adoption of a deprecated release.

Approach. We study the transitive adoption of deprecated releases by analyzing the *dependency tree* of the latest release of client packages. Such dependency trees contain the provider package releases that are directly and transitively adopted by each client in their latest release. To obtain these dependency trees, we run the `npm install` command (to install the client package release and its dependencies) followed by the `npm ls` command (to obtain the dependency tree in a parsable format). The dependency trees represent the resolved provider releases at the time that we run the `npm install` tool (February 2020), whereas the timestamp of the latest client package releases were obtained from our data set (which was collected on May 2019). For simplicity, in this subsection we refer to the "latest client package release" simply as a "client package". Similarly, we refer to the provider releases that are directly and transitively adopted by the client packages as "provider packages". Furthermore, given the total number of client packages to be analyzed (595,052), we draw a statistically representative random sample of size 16,641 (99% confidence level, ±1% confidence interval). In total, the studied dependency trees have 71,663 installed packages (among client and provider packages) and 5,796,506 dependencies.

After obtaining the dependency trees, we estimate the proportion of client packages that transitively adopt at least one deprecated provider release. In addition, to understand the relation between the direct and transitive adoptions of deprecated releases, we calculate how often client packages that directly adopt at least one deprecated provider release also transitively adopt at least one deprecated provider release (and vice-versa).

We also calculate the distribution of the *deepest dependency depth of a deprecated provider release* for each client package. The deepest dependency depth of a deprecated provider release is the largest distance from the client package to any adopted deprecated release in the dependency tree. For instance, suppose that a client package C transitively adopts a deprecated provider release at depths 2 (i.e., the provider of a provider is deprecated) and 3. For client package C, the deepest dependency depth of a deprecated provider is 3.

Finally, we analyze the distribution of the number and the proportion of directly adopted providers that result in the transitive adoption of at least one deprecated release. As an example, suppose that a client package C directly adopts three providers, namely P_1, P_2, and P_3. Suppose that adopting P_1 results in the transitive adoption of deprecated release d_1 and that adopting P_2 results in the transitive adoption of deprecated releases d_2 and d_3. In such hypothetical scenario, the number of directly adopted

Table 6.4: The number of client packages that directly and transitively adopt at least one deprecated provider releases (only client packages that adopt a deprecated provider release are shown).

Client packages		**Directly adopt deprecated releases?**	
		No	*Yes*
Transitively adopt deprecated releases?	*No*	–	1,107 (11%)
	Yes	5,406 (54%)	3,461 (35%)

providers of C that result in the transitive adoption of at least one deprecated release is 2 (P_1 and P_2), while the proportion is 66% (2 out of 3 directly adopted providers).

Observation 6.10) ***54% of all client packages transitively adopt at least one deprecated release.*** For client packages that adopt a deprecated provider release, the majority of such adoptions is exclusively transitive (see Table 6.4). In particular, the number of client packages that adopt only transitive deprecated providers (5,406) is almost 5 times larger than the number of client packages that adopt only direct deprecated providers (1,107).

Observation 6.11) ***In 90% of the cases where a deprecated provider release is adopted, the deepest dependency depth is no larger than 6.*** Figure 6.7 depicts an histogram for the deepest dependency depth of a deprecated provider release. The median value, as represented by the red dashed line, is 4. The 90th percentile, as represented by the blue dash line, is 6. This analysis indicates that a relevant proportion of client packages adopt deprecated releases that are fairly deep in their dependency tree.

Observation 6.12) ***The median number of direct providers that result in the transitive adoption of at least one deprecated release by a client package is 1.*** In other words, for 50% of the client packages, one single direct provider would need to be updated

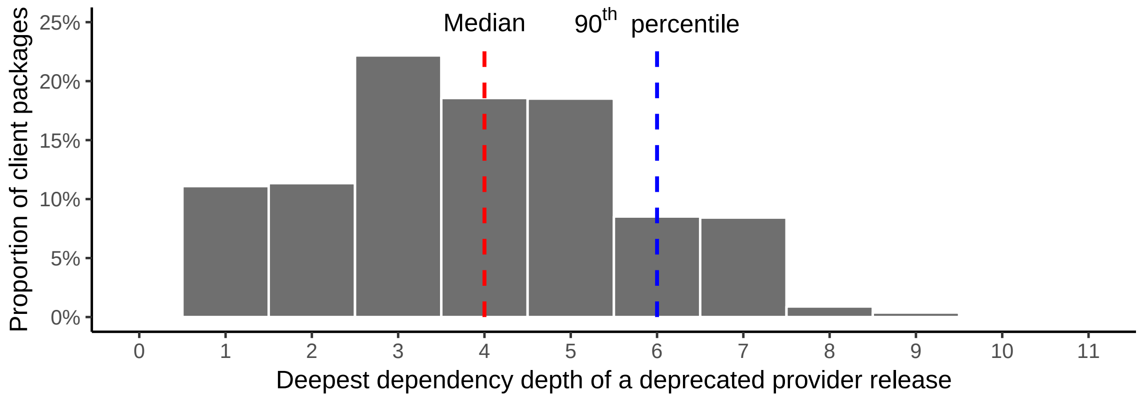


Figure 6.7: Histogram of the deepest dependency depth of a deprecated provider release. The dashed red line represents the median, while the blue one represents the 90th percentile.

Table 6.5: Descriptive statistics for the total number and the proportion of direct providers that result in the transitive adoption of at least one deprecated release.

Providers	**Min.**	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
Total	1.0	1.0	1.0	1.98	2.0	60.0
Proportion	2.0%	16.6%	25.0%	32.5%	42.8%	100.0%

or replaced as an effort to cease the transitive adoption of a deprecated release. In addition, Table 6.5 shows that the median proportion of direct providers that account for the transitive adoption of at least one deprecated release is 25%.

RQ2: How do client packages adopt deprecated releases?

- The direct adoption of deprecated releases is highly skewed, with the top 40 popular deprecated releases accounting for more than half of all deprecated releases adoption.
- All the top 40 popular deprecated releases have a deprecation message that reports a replacement package or release, which eases the migration from such deprecated releases.
- 54% of all client packages transitively adopt at least one deprecated release.
- In 90% of the cases where a deprecated release is adopted, the deepest dependency depth is no larger than 6.
- A median of one in each four providers that are directly adopted result in the transitive adoption of at least one deprecated release.

6.5 Discussion

In this section, we discuss the findings presented in Section 6.4. We divide our discussion in two topics: in Section 6.5.1, we discuss the improvements to the npm deprecation mechanism. In Section 6.5.2, we assess the impact of deprecated releases on client packages.

6.5.1 Improving the deprecation mechanism

Although a small proportion of npm packages make use of the deprecation mechanism (Observation 1), a noteworthy proportion of client packages directly adopt deprecated releases (Observation 6). Therefore, we consider release-level deprecation to be a relevant aspect of the npm ecosystem. Despite such relevance, the deprecation mechanism provided by npm is fairly rudimentary (see Section 6.2 for a description on how the npm deprecation mechanism works). In the following, based on our observations from Section 6.4, we propose specific improvements to the deprecation mechanism.

Implication 6.1) ***The deprecation mechanism should encourage developers to provide more meaningful deprecation messages.*** The rationale for the deprecation is not reported in 36% of the deprecation messages (Observation 5). Informing the rationale behind a deprecation allows client packages to assess the trade-off between the risk of adopting the deprecated release and the effort to migrate away from this release. Therefore, npm should encourage developers to provide better reasons for the deprecation of a release (e.g., by providing standardized deprecation messages based on the five identified rationales for deprecation).

In addition, the replacement release is not reported in approximately half (49%) of the deprecation messages (Observation 5). Reporting a replacement release allows client packages to easily migrate away from the deprecated release, in case they decide to. Hence, we argue that the npm deprecation mechanism should support package developers in informing what the replacement release is (e.g., by automatically detecting the existence of a newer non-deprecated release and, also, by warning developers when the latest package release is being deprecated). In addition, the npm deprecation mechanism should record a deprecation timestamp (to date, this information is not recorded in the npm registry) and a severity level for the deprecation. These information would help client packages on 1) evaluating for how long a release is considered deprecated and 2) assessing the risks of adopting a deprecated release, based on the perspective of the developer of the provider package.

Implication 6.2) ***The deprecation mechanism should interactively prompt for contextual information about the deprecation.*** The provision of contextual information by package developers should be encouraged by enhancing the interaction between the npm deprecation tool and its users. Instead of requiring only the deprecation message and the package release (or range of releases) to be deprecated as parameters, the deprecation tool could interactively prompt its users to input: 1) the rationale behind the deprecation, 2) a pointer to the replacement release, 3) a link to documentation containing instructions on what should be considered when migrating to the replacement release (e.g., the introduced changes in the replacement release), and 4) a deprecation severity level. In addition, to supporting better automated analyses, such contextual information should be recorded in proper fields in the `package.json` file.

Implication 6.3) ***The deprecation mechanism should periodically warn client packages about the adoption of a deprecated provider release.*** Even though all popular deprecated releases report a replacement release in their deprecation message (Observation 8), such releases are still massively adopted by client packages (on their latest release, as of our data collection date) (Observation 7). We theorize that such adoptions happen because the deprecation messages are displayed only when a deprecated provider release is installed. When the provider release is deprecated while it is already installed, the client package does not become aware of such a deprecation. In fact, this issue has been a subject of discussion in several issue reports.[10,11] We argue that npm should provide an easy way for client packages to check the adoption of a deprecated

[10] https://github.com/npm/npm/issues/15536
[11] https://github.com/npm/npm/issues/18023

provider release. Preferably, the deprecation mechanism should proactively (and periodically) warn client packages when a deprecated release is adopted.

6.5.2 Assessing the impact of deprecated releases

Although the migration away from a deprecated release depends on the client package's willingness, we found evidences that such a migration might not be trivial (e.g., by the lack of a replacement release or the need to perform changes to migrate). For this reason, client packages that are willing to migrate away from deprecated releases can benefit from understanding whether a replacement release is available and how difficult such a migration typically is. In turn, client packages that still adopting a deprecated provider release should evaluate the impact and risks associated with such an adoption. In the following, we discuss the risks that client packages can face when adopting a deprecated release, as well as the challenges to migrate away from a deprecated release.

Implication 6.4) ***Client packages should be especially careful about the usage of deprecated releases of partially deprecated packages.*** The deprecation of a release can occur for different reasons. The most common reason for the deprecation of all releases of a package in npm is withdrawal (i.e., terminating the maintenance of a package), whereas the deprecation of one or more specific releases of a package usually occurs due to a defect (Observation 5). Although the usage of provider packages that are no longer maintained should be avoided, the usage of defective provider releases might be considerably more risky and should be addressed with proper attention by client packages (e.g., by migrating away from the deprecated release). Another common reason for deprecation is when a release is superseded. This rationale for deprecation does not indicate any issue that needs to be urgently addressed, however provider packages can have some specific reason (e.g., the deprecated release use some obsolete feature) to communicate a deprecation to client packages instead of simply publishing a newer release. Therefore, client packages should update a superseded provider release whenever it is possible, especially when the newer provider release is backward compatible with the deprecated release. Incompatibility is also identified as a rationale for the deprecation of a package's release, however the deprecation of a release for this reason is significantly less common than for other reasons. Nevertheless, client packages are exposed to incompatibilities in very specific circumstances (i.e., when incompatible versions of two providers are used at once).

Implication 6.5) ***Client packages should be prepared for the lack of replacement releases.*** A total of 31% of the partially deprecated packages do not have a follow-up replacement release, giving client packages no option other than downgrading their adopted provider version (Observation 2). Also, the deprecation mechanism is used primarily (66%) for the deprecation of all releases of a package (Observation 1), which gives client packages no other option besides replacing the whole provider package.

Implication 6.6) ***Client packages should be aware that migrating to a replacement package or release might be a costly operation.*** 80% of the popular deprecated releases belong to fully deprecated packages that only provide a replacement package (in lieu of a replacement release) (Observation 9). The migration to a replacement package requires the adoption of a different provider package, potentially requiring client packages to perform changes in order to cope with a new design implemented by this provider (e.g., new APIs). When a replacement release is available, in 15% of the cases client packages will need to integrate a major provider release (Observation 3), which is assumed to be backward incompatible with the deprecated release, likely requiring client packages to perform changes to integrate the new major provider release. Also, in Observation 9 we show that 91% of the client packages set a version range statement when adopting a deprecated provider release. Even though, such client packages are still adopting a deprecated release, which suggests that a modification of the version range statement might be required to migrate to a replacement release.

Implication 6.7) ***Client packages should be attentive to the transitive adoption of deprecated provider releases.*** More than half (54%) of the client packages in the ecosystem transitively adopt at least one deprecated release. Interestingly, the majority of these client packages (54%) do not directly adopt deprecated releases (Observation 10). Hence, client packages that want to avoid the adoption of deprecated releases by all means should be attentive to their transitive providers. When a deprecated release is transitively adopted, client packages have two main options to cease the adoption:

1) Updating the direct provider that is responsible for the transitive adoption of a deprecated release. This option does not necessarily guarantee that the adoption of the deprecated release will cease, however it serves as a best-effort approach. We verified that the median number of direct providers that result in the transitive adoption of at least one deprecated releases is 1 (Observation 12). This number thus serves as an estimate of how many direct providers would need to be updated for this best-effort approach.

2) Performing workarounds. The deeper a transitively adopted deprecated release is in the dependency tree of a client package, the less control the client package has over such an adoption. Indeed, we verified that the median of the deepest dependency depth of a deprecated provider release is 4 (Observation 11), showing that, in general, client packages need to cope with the transitive adoption of deprecated provider releases that are far down the dependency tree. To manually update transitive adoptions, client packages often rely on workarounds, such as manually modifying build files that are automatically generated and reinstalling all provider packages.[12,13].

6.6 Related work

In this section, we describe prior studies about deprecation in software ecosystems. We initially present related work that discusses how client packages use deprecated APIs and how such clients react to the deprecation of these APIs (Section 6.6.1). Then, we discuss studies that report how often deprecation messages report a replacement API (Section 6.6.2). Finally, we discuss studies that describe the rationale behind the deprecation of APIs (Section 6.6.3). All the presented related work discusses the phenomenon of *API deprecation*, whereas this thesis is the first to present a study about *release deprecation* in a software ecosystem. Therefore, in this section we also compare prior results regarding the API deprecation with our results about release deprecation.

6.6.1 Usage of and migration away from deprecated releases

Henkel and Diwan (2005) discuss that the usage of a certain provider's API by client packages can have an impact on the decision of deprecating this API. The authors argue that provider packages do not want to drive complex changes in the client packages

[12] https://stackoverflow.com/questions/56634474/npm-how-to-update-upgrade-transitive-dependencies

[13] https://stackoverflow.com/questions/15806152/how-do-i-override-nested-npm-dependency-versions

and the decision about the deprecation of an API should consider this assumption. Robbes et al. (2012a) analyzed the deprecation of packages' API in the Pharo ecosystem (for the Smalltalk language) and found that a small proportion (14%) of the deprecated methods triggered a client package reaction (e.g., a method replacement). Sawant et al. (2018c) analyze the reaction of clients to the deprecation of Java APIs and found that a small proportion of the client packages migrate away from the deprecated methods. The authors found that the majority of the client packages do not use any deprecated provider API. In a follow-on study, Sawant et al. (2019) derive the following patterns of reaction to Java API deprecation: deletion of call to deprecated API, replacement by third-party API, replacement by in-house API, and downgrade of API version. Although the authors also found that the majority of the client packages do not migrate away from the deprecated methods, when the migration takes place client packages usually replace the deprecated method with a call to another third-party API. Li et al. (2018) report that 38% of a random sample of 10,000 Android apps are using a deprecated API. Our results show that 27% of all client packages in the ecosystem adopt a deprecated release.

Prior studies show that the rate at which client packages replace an adopted deprecated API is low. In contrast, we found that 85% of the replacement releases in npm are patch and minor releases, which are often implicitly adopted by client packages.

6.6.2 Replacement releases in deprecation messages.

Zhou and Walker (2016) found that 51% of the studied packages in the Maven ecosystem (for the Java language) have a deprecation message that reports a replacement

API. Brito et al. (2016) show that 59.5% of the API elements (types, fields, and methods) of 661 Java projects are deprecated with a message that reports a replacement API element. Ko et al. (2014) reveal that 61% of 260 deprecation messages of eight Java packages have a replacement API. In turn, Li et al. (2018) found that, among a set of 20 Android releases, the median proportion of deprecated APIs that do not report a replacement on the deprecation message is approximately 30%. In a study about API deprecation messages of the top 50 most popular client packages in npm, Nascimento et al. (2020) points that 67% of the deprecation messages report a replacement API. Our results show that 51% of the deprecation messages in npm report a replacement release.

In general, the difference between the proportion of API and release deprecation messages that report a replacement is at most 16%.

6.6.3 Rationale behind deprecation

According to a survey by Sawant et al. (2018a), there are seven rationales for the usage of the deprecation mechanism by client developers in Java. Their study examined deprecation at the method level (i.e., API deprecation). With the exception of one out of the seven stated rationales (namely "old interface encourages bad practices"), all of them have commonalities with the rationales for the deprecation of a release in npm (see Table 6.6). In turn, Sawant et al. (2018b) manually investigate the deprecation messages of 374 Java APIs and propose 12 categories for the rationale of deprecation. Even focusing on Java API deprecation, many of the proposed categories agree with the rationales for the deprecation of npm packages and releases. Mirian et al. (2019) studied the reasons for the deprecation of APIs provided by the Chrome web browser.

The authors identified six different categories for the rationale behind the deprecation of an API. Four out of the six categories are related with the identified rationales for the deprecation of a release in npm. The "inconsistent implementation" and "security" categories by Mirian et al. (2019) are related with defects, the "updated standard" category is related with supersede, and the "removed from standard" category is related with withdrawal. Raemaekers et al. (2014) studied the Maven ecosystem and found that deprecation is rarely used to communicate that a given API has introduced backward incompatible changes (a.k.a breaking changes). Decan et al. (2018) suggest that package developers in npm should deprecate releases that can potentially suffer from vulnerabilities.

There are a number of commonalities between the rationale for deprecating an API and a release. All rationales for release deprecation can also be associated with API deprecation, although the opposite does not hold.

6.7 Threats to validity

In this section, we discuss the threats to the validity of our study about release deprecation in npm. We discuss the threats related to construct validity, internal validity, and external validity.

Construct Validity. The npm registry does not record the date of a release deprecation. Therefore, when our data was collected from the npm registry, we only knew that a given release was deprecated *some* time before our data collection. The lack of knowledge about the deprecation date makes it impossible for us to perform a reliable historical analysis about the adoption of deprecated releases. To mitigate this threat, we do not perform any historical analysis of deprecated releases adoption (e.g., an analysis on

Table 6.6: Comparison between the identified rationales behind API and release deprecation.

Reference	Identified rationales for method deprecation	Identified rationales for release deprecation
Sawant et al. (2018a)	Feature is unnecessary; no longer provide a feature	Withdrawal
	New feature supersedes existing one	Supersede
	Functional issue; non-functional issue	Defect
	Mark as beta	Test
	Old interface encourages bad practices	—
Sawant et al. (2018b)	Redundant methods; renaming of feature	Withdrawal
	Merged to existing method; new feature introduced; separation of concerns	Supersede
	Functional defects; security flaws	Defect
	Temporary feature; dissuade usage	Test
	No dependency support	Incompatibility
	Avoid bad coding practices; design pattern	—
Mirian et al. (2019)	Removed from standard	Withdrawal
	Updated standard	Supersede
	Security; inconsistent implementation	Defect
	Clean experience; never standardized	—

how client packages migrate away from deprecated releases). Rather, we consider how deprecated releases were adopted at the latest client release (i.e., at the snapshot of our data collection). Even though, we can still rely on cases for which the provider release was deprecated at the time of our data collection, but was not deprecated when the client package started adopting this release. Also, when analyzing the adoption of the top 40 popular deprecated releases (Section 6.4.2), we manually analyze any documentation that allows us to estimate the deprecation date. With such an analysis, we can reveal the amount of time that client packages had to migrate away from a popular deprecated release.

In Section 6.4.2, we studied the transitive adoption of deprecated releases. To do so, we installed the client packages using the `npm install` and captured their dependency trees using the `npm ls` tools. We set a 10 minutes timeout for the installation of the client package. Therefore, packages that took more than 10 minutes to be installed were skipped from our analysis.

Internal validity. To understand the rationales behind the deprecation of a release (Section 6.4.1), we manually analyzed a representative sample of unique deprecation messages. We choose to sample *unique deprecation messages* instead of *unique deprecated releases* because each package has a different number of releases and the same package can deprecate a range of releases (perhaps all releases) with the same message. Therefore, by sampling unique deprecation messages instead of unique deprecated releases, we are avoiding a selection bias towards packages with a large number of deprecated releases. Our sample size ensures a confidence level of 95% and a confidence interval of ±5%. Therefore, the reported prevalence for each rationale is bound to the properties of such sample. Also, we might have not sampled messages that refer to rationales that rarely appear. For example, prior studies indicate that vulnerabilities can potentially drive a deprecation (Decan et al., 2018), however our sample did not include any deprecation messages that explicitly refers to a vulnerability (although vulnerabilities would be part of the *defect* category).

External Validity. Our study is limited to data from the npm ecosystem, therefore our results might not be generalized to other ecosystems. Our study is the first to analyze the deprecation phenomenon at the release- and package-levels in a software ecosystem and to provide knowledge about such a phenomena. Nevertheless, we identified that the rationales for the deprecation of releases have commonalities with the deprecation of APIs.

Our study does not have the objective of elucidating general theories about the deprecation phenomenon and further research is needed to provide more comparisons and an eventual generalization. Furthermore, release-level deprecation mechanisms are provided by other package managers ecosystems (e.g., Packagist for the PHP language or Nuget for .NET) and our approach can be replicated in those other ecosystems. Although the deprecation mechanisms of different ecosystems have different characteristics, the learned lessons discussed in Section 6.5 can be useful to reason about release-level deprecation in other ecosystems.[14,15,16]

[14] https://devblogs.microsoft.com/nuget/deprecating-packages-on-nuget-org/

[15] https://www.tomasvotruba.com/blog/2017/07/03/how-to-deprecate-php-package-without-leaving-anyone-behind/

[16] https://api.rubyonrails.org/classes/ActiveSupport/Deprecation.html

6.8 Conclusion

Deprecation is a mechanism employed by software developers to discourage the use of a particular piece of code. Prior empirical studies focused on the deprecation of API elements (e.g., methods and functions) and investigated several topics, such as how frequently deprecated APIs are adopted by clients (Robbes et al., 2012a; Sawant et al., 2018c), the provisions of replacement APIs (Zhou and Walker, 2016; Li et al., 2018; Nascimento et al., 2020), and the rationales behind the deprecation of APIs (Sawant et al., 2018a,b; Mirian et al., 2019). In this chapter, we study deprecation from a software ecosystem perspective, which entails the deprecation of releases. More specifically, we conducted a case study of release deprecation in npm.

To understand the relevance of the deprecation mechanism in npm, we analyzed how often releases are deprecated by provider package developers and the impact of the deprecated releases over the client packages. We found that the rate at which the deprecation mechanism is used by provider packages is small, with approximately 3% of the releases being deprecated. However, 27% of the client packages directly adopt at least one deprecated release and 54% of all client packages transitively adopt at least one deprecated release in their latest release. We assessed the risks brought by the usage of deprecated releases by studying the rationales behind the deprecation. We verified that the deprecation of all releases of a package is usually associated with withdrawals (i.e., terminating the package maintenance) and supersede (i.e., the substitution of a release by another). Also, we verified that the deprecation of one specific package release is usually associated with the presence of defects in that release.

Based on our observations, we conclude that, despite the importance of the deprecation mechanism to the npm ecosystem, such a mechanism is still fairly rudimentary. For instance, to date, there is no simple approach that enable client packages to check whether any installed provider release is deprecated. We propose a series of improvements to the npm deprecation mechanism. We also conclude that it is not straightforward for client package to assess the impact (e.g., risk) of using a deprecated release. For instance, the rationale behind a deprecation is not always provided and client packages can unwittingly adopt deprecated releases by means of transitive dependencies. We propose ways in which client packages can better assess the impact of using deprecated releases.

CHAPTER 7

Conclusions and Future Work

SOFTWARE ecosystems continue to gain popularity as a viable mechanism to enable large scale source code reuse. On the one hand, the large and diverse code base typically associated with those ecosystems supports the fast-paced contemporary software development. On the other hand, the overhead brought by dependency maintenance can become a burden to developers. In this thesis, we propose to leverage data from the npm ecosystem to help practitioners (software developers and package manager owners) to make informed decisions regarding dependency maintenance. Our goal is to study three specific phenomena: downgrades of dependencies, same-day releases, and release deprecation. To accomplish our goal, we 1) mine evolutionary data from the dependencies of the npm ecosystem to understand how and why downgrades are performed; 2) mine historical data from npm package releases and their deployed files to study the phenomenon of same-day releases; 3) mine data from a comprehensive snapshot of the releases and dependencies of the npm ecosystem to understand the prevalence and adoption of deprecated releases. We then perform an empirical analysis of such data to evaluate the driving forces behind these phenomena, as well as their prevalence and impact in the ecosystem.

7.1 Learned lessons

Based on our empirical observations, we propose a set of suggestions to improve dependency maintenance practices in npm. We summarize these suggestions in the form of a set of learned lessons, which are described below:

1. **Monitoring dependency updates to perform better downgrades.** In Chapter 4, we observed that downgrades of implicit updates generally take longer to occur, indicating that the issues that motivate these downgrades are harder to debug or the associated providers with such issues are harder to localize. For this reason, tools to assist the localization of troublesome providers should be provided to client packages. In particular, these tools should facilitate tracing the history of updates of a provider. Emphasizing testing of providers' functionalities can also help in this matter. Also, we observed that some downgrades can lead the adoption of a provider version that is older than the latest working version. Tooling for making developers aware of those cases would be helpful to client packages.

2. **Embracing the need for same-day releases.** In Chapter 5, we observed that, although developed in very rapid fashion, same-day releases introduce relevant and even considerably large-sized changes. Moreover, same-day releases are

adopted relatively fast by client packages, evidencing their importance on the ecosystem. Based on such observations, we argue that popular provider packages should ensure that same-day releases receive proper attention and that, even under such a rapid release pipelines, perform all essential quality assurance checks. We encourage popular npm packages to optimize their release pipelines and strive to cope with the rapid nature of same-day releases. In addition, we note that the documentation of same-day releases by means of release notes should be improved.

3. **Towards a comprehensible deprecation mechanism.** In Chapter 6, we observed that more than one-third of the deprecation messages do not inform the rationale behind the deprecation and almost half do not report a replacement release. Considering the importance of understanding the reasons for a deprecation, as well as whether a replacement release exists, we encourage package manager owners to design deprecation mechanisms that better support developers in providing such information. In addition, the current support of the deprecation mechanism for identifying the adoption of deprecated releases is still limited and should be improved.

7.2 Limitations

One of the limitations of our studies is that we study data exclusive from the npm ecosystem. As a consequence, so we are not able to verify whether our observations hold in other software ecosystems. Another limitation related to the data source is that we restrict our study to the set of dependencies within the npm ecosystem, while external projects can also depend on any npm package. Data collection challenges that are beyond the scope of this thesis need to be overcame first to allow a sound analysis of the dependencies of external projects on npm packages. In particular, it is difficult to accurately trace the state of these external applications, to determine when those external applications perform a release, and which version of the provider package they are using at a certain given time.

7.3 Avenues for future research

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

- *Further research must be carried out to understand how downgrades affect packages throughout the dependency network.* Although we observed that only 19% of the releases with downgrade have at least one client package, downgrades can still transitively impact packages in the ecosystem. For instance, a package a can depend on a package b which, in turn, might depend on a package c. Therefore, package a can transitively depend on some feature of package c. If c is downgraded by b, a will transitively depend on a downgraded version of c. However, in this thesis, we do not investigate the impact of downgrades on transitive dependencies.

- *Further research is necessary to understand why downgrades tend to take a long tome to occur.* We conjecture that either the problem that triggered the downgrade takes long to manifest or tracing a problem back to a certain provider version is not trivial. Also, we conjecture that, due to the controlled nature of explicit updates, it is easier for client packages to identify the provider that is associated with the problem that motivated the downgrade.

- *Further research should be performed to understand the extent to which vulnerability and security advisories might be influencing client developers to downgrade a provider.* Different studies show the relevance of security vulnerabilities to the decision of updating a given provider (Decan et al., 2018; Zerouali et al., 2019a). However, after manually analyzing a representative sample of downgrades (see Section 4.4.1), we did not find any explicit mention of security vulnerabilities. We conjecture that client packages tend to wait for a vulnerability fix instead of performing a downgrade.

- *Further research should be performed to understand the role of automated tools for dependency management.* Although prior work examined whether the adoption of "dependency bots" (i.e., automated tools that interacts with the code to perform early testing of new provider versions) encourage developers to updating their providers (Mirhosseini and Parnin, 2017), it is still not clear how often npm client packages use these tooling and how effective these are in preventing the adoption of problematic provider releases. For example, Dey et al. (2019) studied the commits history of 4,433 npm packages and found that 400 of such packages (less than 10%) have any commit from a dependency bot.

- *Further research should study historical deprecation data.* Due to the lack of deprecation timestamp on data from npm, our study of release and package deprecation does not take historical information into consideration. A feasible way of overcoming this data limitation is to periodically fetch and store data from the npm registry (e.g., daily snapshots), eventually combining each individual measurement on a time series. We leave such complimentary yet important aspect of deprecation for future research endeavours.

Bibliography

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

Adams, B., Bellomo, S., Bird, C., Marshall-Keim, T., Khomh, F., and Moir, K. (2015). The practice and future of release engineering: A roundtable with three release engineers. *Software, IEEE*, 32:42–49.

Adams, B. and McIntosh, S. (2016). Modern Release Engineering in a Nutshell – Why Researchers Should Care. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 78–90.

Barros-Justo, J., Olivieri, D., and Pinciroli, F. (2019). An exploratory study of the standard reuse practice in a medium sized software development firm. *Computer Standards & Interfaces*, 61:137–146.

Bauer, D. F. (1972). Constructing confidence sets using rank statistics. *Journal of the American Statistical Association*, 67(339):687–690.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., and Panichella, S. (2015). How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317.

Bavota, G., Canfora, G., Penta, M. D., Oliveto, R., and Panichella, S. (2013). The evolution of project inter-dependencies in a software ecosystem: The case of Apache. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'13)*, pages 280–289.

Bevacqua, N. (2015). Keeping your npm dependencies immutable. `https://ponyfoo.com/articles/immutable-npm-dependencies`. Accessed: 2018-01-07.

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

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2016). Do developers deprecate APIs with replacement messages? A large-scale analysis on java systems. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 360–369.

Castelluccio, M., An, L., and Khomh, F. (2019). An empirical study of patch uplift in rapid release development pipelines. *Empirical Software Engineering*, 24(5):3008–3044.

Cliff, N. (1996). *Ordinal methods for behavioral data analysis*. Psychology Press, New-York, USA.

Cogo, F. R., Oliva, G. A., and Hassan, A. E. (2019). An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*. Preprint.

Constantinou, E. and Mens, T. (2017). Socio-technical evolution of the ruby ecosystem in github. In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*, pages 34–44.

Cox, J., Bouwers, E., v. Eekelen, M., and Visser, J. (2015). Measuring dependency freshness in software systems. In *Proceedings of the IEEE/ACM 37th International Conference on Software Engineering (ICSE-SEIP'15)*, pages 109–118.

Decan, A. and Mens, T. (2019). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, pages 1–15.

Decan, A., Mens, T., and Claes, M. (2017). An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the24th International Conference on Software Analysis, Evolution, and Reengineering (SANER'2017)*, pages 2–12.

Decan, A., Mens, T., and Constantinou, E. (2018). On the evolution of technical lag in the npm package dependency network. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME'18)*, pages 404–414.

Decan, A., Mens, T., and Constantinou, E. (2018). On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, pages 181–191.

Decan, A., Mens, T., and Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416.

Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. (2017). Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '17)*, pages 2187–2200.

Dey, T., Ma, Y., and Mockus, A. (2019). Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem. In *Proceedings of the 15th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19)*, page 3645.

Dey, T. and Mockus, A. (2018). Are software dependency supply chain metrics useful in predicting change of popularity of npm packages? In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE18)*, page 6669.

Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A., and Ampatzoglou, A. (2018). How do developers fix issues and pay back technical debt in the apache ecosystem? In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*, pages 153–163.

Dodds, K. C. (2015). Why semver ranges are literally the worst? https://blog.kentcdodds.com/why-semver-ranges-are-literally-the-worst-817cdcb09277. Accessed: 2018-01-07.

Draper, P. (2017). Package management: Stop using version ranges. https://www.lucidchart.com/techblog/2017/03/15/package-management-stop-using-version-ranges/. Accessed: 2018-05-28.

Erenkrantz, J. R. (2003). Release management within open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering (OSS'03)*, pages 1–5.

Eski, S. and Buzluca, F. (2011). An empirical study on object-oriented metrics and software evolution in order to reduce testing costs by predicting change-prone classes. In *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 566–571.

Fox, A. (2002). Toward recovery-oriented computing. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB'02)*, pages 873–876.

German, D. M. and Hindle, A. (2005). Measuring fine-grained change in software: Towards modification-aware change metrics. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10–28.

Gkortzis, A., Feitosa, D., and Spinellis, D. (2019). A double-edged sword? Software reuse and potential security vulnerabilities. In *Reuse in the Big Data Era*, pages 187–203.

Gonzalez-Barahona, J. M., Sherwood, P., Robles, G., and Izquierdo, D. (2017). Technical lag in software compilations: Measuring how outdated a software deployment is.

In *Open Source Systems: Towards Robust Practices*, pages 182–192. Springer International Publishing.

Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H. (2000). Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661.

Haenni, N., Lungu, M., Schwarz, N., and Nierstrasz, O. (2014). A quantitative analysis of developer information needs in software ecosystems. In *Proceedings of the 2014 European Conference on Software Architecture Workshops (ECSAW '14)*, pages 1–6.

Hamilton, J. (2007). On designing and deploying internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration Conference (LISA'07)*, pages 1–12.

Hassan, S., Shang, W., and Hassan, A. E. (2017). An empirical study of emergency updates for top android mobile apps. *Empirical Software Engineering*, 22(1):505–546.

Hejderup, J., v. Deursen, A., and Gousios, G. (2018). Software ecosystem call graph for dependency management. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER'18)*, pages 101–104.

Henkel, J. and Diwan, A. (2005). Catchup! Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05).*, pages 274–283.

Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A., and Ducasse, S. (2018). How do developers react to API evolution? A large-scale empirical study. *Software Quality Journal*, 26(1):161–191.

Ihara, A., Fujibayashi, D., Suwa, H., Kula, R. G., and Matsumoto, K. (2017). *Understanding when to adopt a library: A case study on ASF projects*, pages 128–138. Springer International Publishing, Cham.

Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E. K., Winter, C., and Murphy-Hill, E. (2018). Advantages and disadvantages of a monolithic repository: A case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'18)*, pages 225–234.

Kajko-Mattsson, M. and Yulong, F. (2005). Outlining a model of a release management process. *Journal of Integrated Design and Process Science*, 9(4):13–25.

Kerzazi, N. and Adams, B. (2016). Botched Releases: Do We Need to Roll Back? Empirical Study on a Commercial Web App. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 574–583.

Khomh, F., Adams, B., Dhaliwal, T., and Zou, Y. (2015). Understanding the impact of rapid releases on software quality. *Empirical Software Engineering*, 20(2):336–373.

Kikas, R., Gousios, G., Dumas, M., and Pfahl, D. (2017). Structure and evolution of package dependency networks. In *IEEE International Working Conference on Mining Software Repositories (MSR'17)*, pages 102–112.

Ko, D., Ma, K., Park, S., Kim, S., Kim, D., and Traon, Y. L. (2014). API document quality for resolving deprecated APIs. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC'14)*, volume 2, pages 27–30.

Kula, E., Rastogi, A., Huijgens, H., Deursen, A. v., and Gousios, G. (2019). Releasing fast and slow: An exploratory case study at ing. In *Proceedings of the 27th ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'19), pages 785–795.

Kula, R. G., German, D. M., Ishio, T., and Inoue, K. (2015). Trusting a library: A study of the latency to adopt the latest Maven release. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, (SANER'15)*, pages 520–524.

Kula, R. G., German, D. M., Ishio, T., Ouni, A., and Inoue, K. (2017a). An exploratory study on library aging by monitoring client usage in a software ecosystem. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'17)*, pages 407–411.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2017b). Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, pages 1–34.

Lahtela, A. and Jäntti, M. (2011). Challenges and problems in release management process: A case study. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Service Science (ICSESS'11)*, pages 10–13.

Lex, A., Gehlenborg, N., Strobelt, H., Vuillemot, R., and Pfister, H. (2014). Upset: Visualization of intersecting sets. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1983–1992.

Li, L., Gao, J., Bissyandé, T. F., Ma, L., Xia, X., and Klein, J. (2018). Characterising deprecated android APIs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, pages 254–264.

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

Lin, D., Bezemer, C. P., and Hassan, A. E. (2017). Studying the urgent updates of popular games on the Steam platform. *Empirical Software Engineering*, 22(4):2095–2126.

Manikas, K. (2016). Revisiting software ecosystems research. *Journal of Systems and Software*, 117(C):84–103.

Manikas, K. and Hansen, K. M. (2013). Software ecosystems – a systematic literature review. *Journal of Systems and Software*, 86(5):1294 – 1306.

Mäntylä, M. V., Adams, B., Khomh, F., Engström, E., and Petersen, K. (2015). On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425.

McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of API stability and adoption in the Android ecosystem. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'13)*, pages 70–79.

McIlroy, M. D. (1969). "Mass produced" software components. In Naur, P. and Randell, B., editors, *Software Engineering*, pages 138–155, Brussels. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

Mezzetti, G., Møller, A., and Torp, M. T. (2018). Type regression testing to detect breaking changes in node.js libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*.

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

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

Mirian, A., Bhagat, N., Sadowski, C., Porter Felt, A., Savage, S., and M. Voelker, G. (2019). Web feature deprecation: A case study for Chrome. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'19)*, pages 302–311.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., and Canfora, G. (2014). Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, pages 484–495.

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 181–190.

Mujahid, S., Abdalkareem, R., Shihab, E., and McIntosh, S. (2020). Using Others' Tests to Identify Breaking Updates. In *Proceedings of the International Conference on Mining Software Repositories (MSR'20)*, page To appear.

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

Nascimento, R., Brito, A., Hora, A., and Figueiredo, E. (2020). Javascript API deprecation in the wild: A first assessment. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'2020)*.

Pfretzschner, B. and ben Othmane, L. (2017). Identification of dependency-based attacks on node.js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES'17)*, pages 1–6.

Raemaekers, S., van Deursen, A., and Visser, J. (2014). Semantic versioning versus breaking changes: A study of the Maven repository. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, pages 215–224.

Robbes, R., Lungu, M., and Röthlisberger, D. (2012a). How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 1–11.

Robbes, R., Lungu, M., and Rothlisberger, D. (2012b). How do developers react to API deprecation? The case of Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 1–11.

Romano, J., Kromrey, J., Coraggio, J., and Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In *Annual Meeting of the Florida Association of Institutional Research*.

Ruiz, I. J. M., Nagappan, M., Adams, B., Berger, T., Dienst, S., and Hassan, A. E. (2016). Analyzing ad library updates in android apps. *IEEE Software*, 33(2):74–80.

Salza, P., Palomba, F., Di Nucci, D., D'Uva, C., De Lucia, A., and Ferrucci, F. (2018). Do developers update third-party libraries in mobile apps? In *Proceedings of the 26th Conference on Program Comprehension (ICPC'18)*, pages 255–265.

Sawant, A. A., Aniche, M., van Deursen, A., and Bacchelli, A. (2018a). Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, pages 561–571.

Sawant, A. A., Huang, G., Vilen, G., Stojkovski, S., and Bacchelli, A. (2018b). Why are features deprecated? An investigation into the motivation behind deprecation. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME'18)*, pages 13–24.

Sawant, A. A., Robbes, R., and Bacchelli, A. (2018c). On the reaction to deprecation of clients of 4+1 popular java APIs and the JDK. *Empirical Software Engineering*, 23(4):2158–2197.

Sawant, A. A., Robbes, R., and Bacchelli, A. (2019). To react, or not to react: Patterns of reaction to API deprecation. *Empirical Software Engineering*, 24(6):3824–3870.

Schober, P., Boer, C., and A. Schwarte, L. (2018). Correlation coefficients: Appropriate use and interpretation. *Anesthesia & Analgesia*, 126(5).

Serebrenik, A. and Mens, T. (2015). Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW'15)*, pages 1–6.

Shihab, E., Mockus, A., Kamei, Y., Adams, B., and Hassan, A. E. (2011). High-impact defects: A study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*, pages 300–310.

Standish, T. A. (1984). An essay on software reuse. *IEEE Transactions on Software Engineering*, (5):494–497.

Stol, K.-J., Ralph, P., and Fitzgerald, B. (2016). Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 120–131.

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE'76)*, pages 492–497.

Van der Hoek, A. and Wolf, A. L. (2003). Software release management for component-based software. *Software - Practice and Experience*, 33(1):77–98.

Wasowski, A. (2020). Dependency bugs: The dark side of variability, reuse and modularity. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS'20)*, pages 1–3.

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

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*, pages 1–10.

Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., and Ihara, A. (2018). Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, pages 559–563.

Zerouali, A., Cosentino, V., Mens, T., Robles, G., and Gonzalez-Barahona, J. M. (2019a). On the impact of outdated and vulnerable javascript packages in docker images. In *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pages 619–623.

Zerouali, A., Mens, T., Gonzalez-Barahona, J., Decan, A., Constantinou, E., and Robles, G. (2019b). A formal framework for measuring technical lag in component repositories and its application to npm. *Journal of Software: Evolution and Process.*

Zerouali, A., Mens, T., Robles, G., and Gonzalez-Barahona, J. M. (2019c). On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, pages 491–501.

Zhou, J. and Walker, R. J. (2016). API deprecation: A retrospective analysis and detection method for code examples on the web. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE'16)*, pages 266–277.

Zimmermann, M., Staicu, C.-A., Tenny, C., and Pradel, M. (2019). Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*, pages 995–1010.

Appendices

A Changes introduced in same-day releases

In Table A.1 of this appendix, we provide a detailed description of each of the identified notable changes in the same-day release notes. See Section 5.4 for a detailed description of our manual analysis of same-day release notes.

Table A.1: Detailed description and example quotes of the notable changes in same-day releases.

Category	Description	Example quotes
Change business logic (18%)	The package's behaviour is modified so that the correct business rules are followed.	"Start webserver and browsers after preprocessing completed" (release note of karma@0.12.35) "changing attributes on '<option>' elements will now have the correct consequences" (release note of jsdom@1.3.1) "Fix: 'no-return-wrap' rule missing from index.js" (release note of eslint-plugin-promise@3.2.1) "'inspect()' checksum of empty file is now 'null'." (release note of fs-jetpack@0.5.2)
Correctly integrate provider (8%)	A provider package is correctly integrated into the client package.	"properly add pnp plugins" (release note of poi@12.4.2) "When ipfs-api runs inside Electron (...) the request logging is breaks [sic] the lib. It will be better to check whether res.req is set instead of relying on detect-node" (pull-request of ipfs-api@22.2.1) "Corrected CSSO API usage" (commit #df6f48 of package svgo@0.6.6)

Category	**Description**	**Example quotes**
Avoid crashing (7%)	A change is performed to avoid unrecoverable failures.	“Fixed: ‘declaration-block-no-ignored-properties‘ now longer crashes on nested rules.” (release note of stylelint@6.3.1) “Fix bug where using ‘Object.create(null)‘ as a rejection reason would crash bluebird” (release note of bluebird@2.5.2)
Refactoring (7%)	A refactoring operation is performed.	“Tidy up build process” (release note of buble@0.4.23) “rename ’startBytes’ to ’inBytes’ and ’endBytes’ to ’outBytes”’ (release note of svgo@0.1.2)
Add new feature (6%)	A new feature is made available in the release.	“add a clean command to wipe out local dirs” (release note of gatsby@2.1.8) “src: add SET_SIZEOF and SET_ALIGNOF macros” (release note of ref@0.3.5)
Change configuration option (5%)	The set of values for a configuration option is modified.	“add altNames + csrConfigFile options to createCertificate” (release note of pem@1.14.1) “Fixed validation of ‘webpack.styles‘ config to allow it to be set to ‘false‘” (release note of nwb@0.16.2) “Add "–host" flag to ‘firebase experimental:functions:shell‘ (...)” (release note of gatsby@2.2.1)

Category	**Description**	**Example quotes**
Fix UI error (5%)	A change is performed to fix a failure that causes a User Interface (IU) malfunctioning.	"The previous Chrome deprecation fixes broke spatial positioning in Safari" (release note of howler@2.0.12) "UI now responds to touch events, and works on mobile devicesUI now responds to touch events, and works on mobile devices" (release note of react-select@0.2.12)
Revert previous release (5%)	The release is published in an explicit attempt to revert a change introduced in the previous release	"Incorrectly published the previous version from the wrong branch. That version doesn't contain all the work that is [sic] was supposed to. You shold [sic] use this version instead." (release note of bookshelf@0.14.1) "please ignore 0.20.0 (...)" (release note of rework@0.20.1) "Fix package publish" (release note of vue-awesome@3.0.2)
Improve build (4%)	A change to improve the build process is introduced	"Remove source maps from build" (release note of zoroaster@3.6.1) "Tidy up build processTidy up build process" (release note of buble@0.4.23) "Reduce size of distributed build" (release note of radium@0.12.2)

Category	Description	Example quotes
Improve documentation (4%)	A change to improve the package's documentation is introduced	"README.md link to "popular projects" presets doesn't work" (closed issue #285 of package standard-changelog@1.0.17) "Add note about API permissions (...) Without the correct permissions you'll get an error (...) Ideally the plugin would check for permissions and raise a more helpful error, but a note in the readme should help until then" (closed issue report #13037 of package gatsby@2.3.6)
Version control chore (4%)	A VCS operation (such as a branch merging) is performed.	"Merged _diffDeep fix" (release note of package config@0.4.20) "fix(release): cherry-pick the release commit to master on success" (commit #6830e of package sweetalert2@7.26.19) "Merged a branch that needed to be merged" (release note of package saasdoc@1.3.1)
Fix bug caused by provider (3%)	A bug introduced by a provider package is fixed.	"update bili [one of the package's dependencies] to fix a transpilation bug" (commit #5e4c8 of package rollup-plugin-postcss@1.2.1) "It seems a rollup-pluginutils [one of the package's dependencies] failure" (issue report #303 of rollup-plugin-commonjs@9.3.2)

Category	**Description**	**Example quotes**
Fix typo (3%)	An incidental typing error is fixed.	"Skip comments between imports" (pull request #164 of package postcss-import@8.0.2) "Fix version number in CHANGELOG.md" (release note of package ecstatic@2.2.1)
Optimize performance (3%)	A performance optimization change is introduced.	"improve json parser speed" (release notes of body-parser@1.1.2) "(...) overhaul to LiveReload backend to make it faster and more robust" (release note of package budo@10.0.3)
Update provider package (3%)	A provider package is updated to a newer version.	"Update libphonenumber@8.10.1" (release note of package google-libphonenumber@3.2.1) "Update Tippex to ∧2.1.1" (release note of package rollup-plugin-typescript@0.7.3)
Address failed bug fix (2.5%)	A failed attempt to fix a known bug drives a new attempt, which results in a release.	"The bug it still there after updating to 1.1.9, please take care to do tests before closing issues (...)" (issue #1093 of package ng2-bootstrap@1.1.14) "Same memory leak fixes as 0.4.1, properly applied to batch() operations too" (release note of package leveldown@0.4.2)

Category	Description	Example quotes
Deprecated package's feature (2.5%)	A package's feature is deprecated.	"inform user that 'start-selenium' is deprecated" (release notes of selenium-standalone@3.0.3) "Deprecated - When using '<include>' with body content, nested body content is now passed in as 'String' property named 'body' (...)" (release note of package marko@2.0.4)
Add missing file (2%)	The release add an important file that was missing on the previous release.	"include missing file in publishing" (release note of conventional-recommended-bump@2.0.1) "Fix bug where deploying functions resulted in the error message "npm ERR! missing script: build" (...)" (release note of package firebase-tools@3.17.1)
Add newer runtime support (1.5%)	A change for supporting a new runtime environment (Node) is performed	"Support Node.js 0.10.16" (release note of package ioredis@1.3.6)
Add provider package (1.5%)	A new provider package is introduced	"add 'ignore' dep" (release note of package now@0.24.1) "Use rollup-babelUse rollup-babel" (release note of package sorcery@0.6.5)

Category	**Description**	**Example quotes**
Fix dependency failure (1.5%)	A change to cope with a failure in a dependency loading is introduced.	“When installing the last version via NPM (...) signal-exit module seems to be missing.” (pull-request #3186 of package stylelint@9.1.1) “Fix AMD-related failures first appearing in v3.5.1” (release note of package mocha@3.5.2)
Fix lint error (1.5%)	A change to adjust static analysis error is introduced.	“fix linting errors” (release note of package svg-sprite-loader@3.7.3)
Add backward compatibility (1%)	A change to add backward compatibility support for a feature is introduced.	“Support prefixing for old flexbox implementations” (release note of package radium@0.12.2)
Add hardware support (1%)	A change to add a new hardware support is introduced.	“There were a few different issues with these "helpers": ‘data.key‘ didn’t account for international keyboards (...)” (release note of package slate-react@0.5.4)

Category	**Description**	**Example quotes**
Address security vulnerability (1%)	A change to address a security vulnerability is introduced.	"Fixed more security vulnerabilities" (release note of package mathjs@3.11.1)
Fix memory leak (1%)	A change to address a memory leak defect is introduced.	"Fix memory leak caused when passing String objects in as keys and values (...)" (release note of package leveldown@0.4.2)
Prevent error (1%)	A change is introduced to prevent some defect.	"Prevent error when destructured [sic] path is not in known globals" (release note of package eslint-plugin-ember@5.0.1)

B Top 40 popular deprecated releases

In this appendix, we provide information regarding the popular (top-40 frequently used) deprecated releases (see Section 6.4.2 to more details about the popular deprecated releases). Table B.2 shows the popular deprecated releases in npm at the time of our data collection. The column "Replacement package or release" shows the name of the replacement package, when the deprecation message provides a replacement package (e.g., babel-preset-env), or the replacement release, when the deprecation message provides a replacement release (e.g., gulp@>= 4).

Table B.2: Popular deprecated releases and their replacement package or release.

Package name	Deprecated release	Replacement package or release	Deprecation date estimate	Evidence for date estimate
babel-preset-es2015	6.24.1, 6.18.0, 6.9.0, 6.6.0, 6.22.0, 6.3.13, 6.14.0, 6.24.0, 6.16.0, 6.13.2, 6.5.0	babel-preset-env	December, 2016	Documentation reports[1] that deprecation of babel-preset-es2015 occurred after release 1.0.0 of babel-preset-env (December 9, 2016).
istanbul	0.4.5, 0.3.22, 0.4.2, 0.4.3, 0.4.4	nyc	May, 2015	Istanbul and nyc packages were merged when nyc released version 2.0.0.[2]

[1] https://github.com/babel/babel-preset-env/pull/65
[2] https://github.com/istanbuljs/nyc/issues/524#issuecomment-280979372

Package name	Deprecated release	Replacement package or release	Deprecation date estimate	Evidence for date estimate
gulp-util	3.0.8, 3.0.7, 3.0.6	vinyl, replace-ext, ansi-colors, date-format, fancy-log, lodash.template, minimist, beeper, through2, multi-pipe, list-stream, plugin-error	December, 2017	Deprecation message reports documentation with deprecation date information.[3]
babel	6.23.0, 6.5.2	babel-cli	February, 2017	Deprecation message reports that deprecation occurs on release “6.x”. We assume the date of the latest release on 6.x branch (6.23.0).

[3]https://medium.com/gulpjs/gulp-util-ca3b1f9f9ac5

Package name	Deprecated release	Replacement package or release	Deprecation date estimate	Evidence for date estimate
react-dom	16.2.0	react-dom@>= 16.2.1	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[4]
core-js	2.4.1, 2.5.1	core-js@latest or core-js@>= 3	February, 2019	Deprecation message reports that deprecation occurs at release 2.6.5.
coffee-script	1.10.0, 1.6.3, 1.7.1, 1.8.0	coffeescript	February, 2017	Latest package release and release 1.0.0 of replacement package.
react-dom	16.4.1	react-dom@>= 16.4.2	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[5]

[4] https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html
[5] https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html

Package name	Deprecated release	Replacement package or release	Deprecation date estimate	Evidence for date estimate
core-js	2.5.7	core-js@latest or core-js@>=3	February, 2019	Deprecation message reports that deprecation occurs at release 2.6.5
jade	1.11.0	pug	April, 2016	jade becomes pug on version 2.0.0, released on April 1, 2016.[6,7]
react-dom	16.3.2	react-dom@16.3.3	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[8]
validate-commit-msg	2.14.0	commitlint	October, 2017	Latest package release and release 4.2.0 (first release) of replacement package.

[6]https://www.npmjs.com/package/pug
[7]https://github.com/pugjs/pug/commit/ab26404b880a1db0b44269354d11d9c55ab4862f
[8]https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html

Package name	Deprecated release	Replacement package or release	Deprecation date estimate	Evidence for date estimate
babel-preset-es2017	6.24.1	babel-preset-env	September, 2017	Latest package release.
node-uuid	1.4.7	uuid	March, 2017	Latest package release.
core-js	2.5.3	core-js@latest or core-js@>=3	February, 2019	Deprecation message reports that deprecation occurs at release 2.6.5.
rollup-watch	4.3.1	rollup	August, 2017	Documentation reports that rollup-watch package was deprecated at release 0.46.0 of rollup package (August 11, 2017).[9]
isparta-loader	2.0.0	istanbul-instrumenter-loader	November, 2011	Latest package release.

[9] https://github.com/rollup/rollup-watch

Package name	Deprecated release	Replacement package or release	Deprecation date estimate	Evidence for date estimate
gulp-minify-css	1.2.4	gulp-clean-css	December, 2015	Package has deprecation commit.[10]
react-dom	16.0.0	react-dom@>= 16.0.1	August, 2018	Vulnerability that drove deprecation was reported on August 01, 2018.[11]

[10] https://github.com/scniro/gulp-clean-css/commit/438a4faf27f134c30e3e94024a83951c21fdc5cc

[11] https://reactjs.org/blog/2018/08/01/react-v-16-4-2.html