Lost in Zero Space – An Empirical Comparison of 0.y.z Releases in Software Package Distributions

Alexandre Decan[a,*], Tom Mens[a]

[a]*Software Engineering Lab, University of Mons, Avenue Maistriau 15, Mons 7000, Belgium*

Abstract

Distributions of open source software packages dedicated to specific programming languages facilitate software development by allowing software projects to depend on the functionality provided by such reusable packages. The health of a software project can be affected by the maturity of the packages on which it depends. The version numbers of the used package releases provide an indication of their maturity. Packages with a 0.y.z version number are commonly assumed to be under initial development, suggesting that they are likely to be less stable, and depending on them may be considered as less healthy.

In this paper, we empirically study, for four open source package distributions (Cargo, npm, Packagist and RubyGems) to which extent 0.y.z package releases and ≥1.0.0 package releases behave differently. We quantify the prevalence of 0.y.z releases, we explore how long packages remain in the initial development stage, we compare the update frequency of 0.y.z and ≥1.0.0 package releases, we study how often 0.y.z releases are required by other packages, we assess whether semantic versioning is respected for dependencies towards them, and we compare some characteristics of 0.y.z and ≥1.0.0 package repositories hosted on GitHub. Among others, we observe that package distributions are more permissive than what semantic versioning dictates for 0.y.z releases, and that many of the 0.y.z releases can actually be regarded as mature packages. As a consequence, the version number does not provide a good indication of the maturity of a package release.

Keywords: software package distribution, software reuse, software library, version management, semantic versioning, software health

1. Introduction

Open source software development embraces the principles of software reuse, through the availability of software package distributions dedicated to specific

*Corresponding author

Email addresses: alexandre.decan@umons.ac.be (Alexandre Decan), tom.mens@umons.ac.be (Tom Mens)

programming languages (e.g., Cargo for Rust, npm for JavaScript, RubyGems for Ruby, and Packagist for PHP). As is the case for any software system, the reusable packages in such distributions can have different levels of maturity. In order for a mature software project to be considered healthy, it should avoid depending on unstable and immature reusable packages that are still in their initial development phase.

A common convention for packages to reflect this maturity in their version number major.minor.patch is to set the major version component to 1 as soon as they reach their first stable release: *"in open source, tagging the 1.0.0 release is comparable to shipping a final product"* [1], *"1.0.0 indicates some degree of production readiness"* [2]. Packages that are under initial development assign a major version number 0 to their releases, conveying that the software is still incomplete and remains work in progress. A 0.y.z version number can therefore be seen as a signal to treat the package differently than a $\geq$1.0.0 version. A JavaScript developer states this as follows: *"it means that a project cannot be trusted. It would be unwise for a business-critical application to have a dependency that is young and prone to significant changes at any time. It also indicates that the dependency project simply isn't done and might not be ready for use."* [1].

If we assume that this convention is followed, it would be advisable for $\geq$1.0.0 packages to not depend on such 0.y.z packages. Similarly, it would be advisable for 0.y.z packages to quickly reach a $\geq$1.0.0 release in order to allow other packages to depend on a stable and mature release. As witnessed by a Rust developer, *"some reasonably mature crates are still at version 0.y.z or depend on other crates that are at 0.y.z. This is seen as a bad thing and usually results in a few GitHub issues pushing the authors of those crates to change the version to 1.0."* [3] In practice, however, many packages remain *stuck in zero version space*. There seems to be a psychological barrier associated to crossing the 1.0.0 version that package maintainers may associate with additional responsibilities.

Some versioning policies explicitly materialise the differences between 0.y.z and $\geq$1.0.0 versions. Consider for example semver (semantic versioning) [4], a common versioning policy in package distributions [5], dictating how version numbers should be incremented w.r.t. backward compatibility. This policy distinguishes 0.y.z from $\geq$1.0.0 versions in terms of maturity, release cycle and stability. For example, semver enables to assess the backwards compatibility of a $\geq$1.0.0 release based on its version number by distinguishing between incompatible changes (i.e., an increment in the major component of the version number) and compatible changes (i.e., an increment in the minor or patch component). For 0.y.z releases on the other hand, semver considers that *"anything may change at any time"*. This could be problematic for developers of dependent packages, since they need to find other ways to assess the compatibility of such releases. They could also decide to stay on the safe side by preventing their installation, implying they will not benefit from the bug and security fixes provided by the new version.

The specific rules for 0.y.z versions are sometimes considered disruptive[1] and counter-intuitive.[2] The term *magic zero* reflects this different semantics for 0.y.z versions. The confusion around magic zero notably led the maintainers of npm to recommend package developers to avoid using 0.y.z version numbers and start from version 1.0.0 "*since the semver spec is weirdly magical about 0.x.y versions, and we cannot ever hope to get everyone to believe what the correct interpretation of 0.x versions are.*"[3] This also gave rise to the satirical *ZeroVer* versioning policy, stating that "*your software's major version should never exceed the first and most important number in computing: zero*".[4]

The goal of this article is to assess quantitatively to what extent package developers in different package distributions take into account such differences in package releases. To reach this goal, we study the following research questions in Cargo, npm, Packagist and RubyGems, four package distributions that are known to adhere to semver [6]:

RQ_1**:** *How prevalent are 0.y.z packages?* We observe in all studied distributions that a high proportion of packages did not yet reach a $\geq$1.0.0 release.

RQ_2**:** *Do packages get stuck in the zero version space?* The overwhelming majority of packages never traversed the 1.0.0 barrier, and of those that did, more than one out of five took more than a year to do so.

RQ_3**:** *Are 0.y.z releases published more frequently than $\geq$1.0.0 releases?* Although a statistical difference could be observed, this difference was small to negligible in each studied distribution.

RQ_4**:** *Are 0.y.z package releases required by other packages?* This was observed to be a frequent phenomenon for each considered package distribution.

RQ_5**:** *How permissive are the dependency constraints towards required 0.y.z packages?* Most dependency constraints towards 0.y.z packages accept new patches, making them more permissive than what semver recommends. According to semver, *"Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable"* [4].

RQ_6**:** *Do GitHub repositories for 0.y.z packages have different characteristics?* GitHub repositories associated to 0.y.z packages were found to have slightly less stars, forks, contributors, open issues and dependent repositories, and to be slightly smaller than repositories for $\geq$1.0.0 packages.

This paper builds further upon previous work [7] in which we empirically analysed the prevalence of 0.y.z releases in Cargo, npm and Packagist, and observed some preliminary evidences of the (lack of) differences between 0.y.z and

[1] `https://github.com/semver/semver/issues/221`
[2] `https://github.com/npm/node-semver/issues/79`
[3] `https://github.com/npm/init-package-json/commit/363a17bc31bf653`
[4] `https://0ver.org`

≥1.0.0 packages. The current paper extends this study by including a fourth package distribution (namely RubyGems) and one year of extra data for the other three distributions, accounting for more than 500K additional packages for the empirical study. We carry out deeper analyses to complement the preliminary insights we got, such as studying the activity of 0.y.z and ≥1.0.0 packages (RQ_1), the time it takes to cross the magic 1.0.0 barrier (RQ_2), the evolution of the release frequency of 0.y.z and ≥1.0.0 releases (RQ_3) and the number of dependent packages (RQ_4) for 0.y.z and ≥1.0.0 packages, and the evolution of dependency constraints in 0.y.z releases (RQ_5). We also added an entirely novel research question on the characteristics of 0.y.z and ≥1.0.0 package repositories on GitHub (RQ_6). We complement our discussions with anecdotal evidence from developers, and an analysis of the version numbers used for initial package releases.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces the research methodology. Section 4 empirically studies the research questions and presents the main findings. Section 5 discusses the results and Section 6 presents the threats to validity of the research, and Section 7 concludes.

2. Related Work

The staged software life cycle model [8] suggests that the *initial development* stage of software should be distinguished from its subsequent evolution: "*During initial development, engineers build the first functioning version of the software from scratch to satisfy initial requirements.*" This staged model has been studied in the context of open source software projects by Capiluppi et al. [9], who observed that "*many FLOSS projects could be argued to never have left this [initial development] stage*". Fernandez et al. [10] observed that the software evolution laws apply well to open source projects having achieved maturity. They confirm, however, that "*many projects do not pass the initial development stage.*" Similarly, Costa et al. [11] observed that 44 out of 60 evolving academic software projects (i.e., 73%) are either in initial development or closedown stage.

The typical way to distinguish initial software development releases from stable ones is by resorting to some kind of versioning scheme. For software libraries, the most common approach appears to be to use some variant of major.minor.patch version numbers. The expressiveness of such a version numbering has been accused of being too limited [12]: "*Especially if component developers need to assign version numbers to their components manually and do not have proper instructions that define which changes in what level of contract conduct a new version, those version numbers at most rest for marketing use and do not ensure compatibility between different components.*" The semver policy [4] was introduced in an attempt to address such issues, and to provide a partial solution to the "dependency hell" that developers face when reusing software packages. It conveys a meaning to the major.minor.patch version number, assuming that the reusable package has a public API: "*Bug fixes not affecting the API*

increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version."

The use of semver is quite common for package distributions [5]. Wittern et al. [13] studied the evolution of a subset of npm packages, analysing characteristics such as their dependencies, update frequency, popularity, and version numbering. They found that package maintainers adopt numbering schemes that may not fully adhere to the semantic versioning principle; and that a large number of package maintainers are reluctant to ever release a version 1.0.0. Raemaekers et al. [14] investigated the semver-compliance in 22K Java libraries in Maven over a seven-year time period. They found that breaking changes appear in one third of all releases, including minor releases and patches, implying that semver is not a common practice in Maven. Because of this, many packages use strict dependency constraints and package maintainers avoid upgrading to newer versions of dependent packages. Decan et al. [15] studied the use of package dependency constraints in npm, CRAN and RubyGems. They observed that, while strict dependency constraints prevent backward incompatibility issues, they also increase the risk of having dependency conflicts, outdated dependencies and missing important updates. Decan et al. [6] studied semver-compliance in four evolving package distributions (Cargo, npm, Packagist and RubyGems). They observed that these distributions are becoming more semver-compliant over time, and that package distributions use specific notations, characteristics, maturity and policies that play an important role in the degree of such compliance.

Bogart et al. [16] qualitatively compared npm, CRAN and Eclipse, to understand the impact of community values, tools and policies on breaking changes. They identified two main types of mitigation strategies to reduce the exposure to changes in dependencies: limiting the number of dependencies, and depending only on "trusted packages". They also found that policies and practices may diverge when policies are perceived to be misaligned with the community values and the platform mechanisms. They confirmed this in a follow-up qualitative study about values and practices in 18 software package distributions, on the basis of a survey involving more than 2,000 developers [5]. Different package distributions were found to have different priorities and make different value trade-offs. Their results show that relationships between values and practices are not always straightforward.

3. Data Extraction

In previous work we studied the use of semver in four package distributions (Cargo, npm, Packagist and RubyGems) [6] and observed that these distributions where mostly semver-compliant. We also observed intriguing differences between how pre-1.0.0 and post-1.0.0 dependency constraints were being used. This triggered us to conduct the current in-depth study on the presence and use of 0.y.z package releases in these package distributions.

To analyze the considered package distributions, we rely on version 1.6.0 of libraries.io Open Source Repository and Dependency Metadata [17], released in January 2020. This dataset contains, among others, the metadata of packages in Cargo, npm, Packagist and RubyGems. These metadata include all package releases, their version number, their release date, their dependencies including the target package, the dependency constraint and the scope of the dependency (e.g., runtime, test, ...). The dataset also contains various metadata for the git repositories related to these packages, such as the address of the repository, the number of contributors, the number of forks, stars, etc.

For each package distribution, we consider all packages and all their releases, except for the pre-release versions (such as 2.1.3-alpha, 0.5.0-beta or 3.0.0-rc) that are known to be "*unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version*" [4]. For each package release, we consider only dependencies to other packages within the same distribution, i.e., we ignore dependencies targeting external sources (e.g., websites or git repositories). When declaring dependencies, a package maintainer can specify the purpose of the dependency (e.g., it is needed to execute, develop or test the package). We excluded dependencies that are only needed to test or develop a package because not all considered packages make use of them, and not every package declares a complete and reliable list of such dependencies. We therefore consider only those dependencies that are required to install and execute the package, and hence more accurately reflect what is needed to actually use the package. In the package distributions we analyzed, these dependencies are either marked as "runtime" or "normal".

To reduce noise in the dataset, we manually inspected and removed outlier packages with clearly deviating and undesirable behaviour. For Packagist we excluded 23 of the most "active" packages (and their associated 1.2K releases) that were created and published to promote illegal download services (e.g., 123movies/watch-pacific-rim-uprising-online-free-123putlocker). For npm we excluded around 23K packages (and their associated 51K releases) that were purposefully created by malevolent developers abusing the API of the npm package manager. These are either packages whose main purpose is to depend on a very large number of other packages (e.g., npm-gen-all) or replications and variations of existing packages (e.g., npmdoc-*, npmtest-*, *-cdn, etc.) Most of them are no longer available through npm.

Table 1: Characteristics of the curated dataset.

distribution	**created**	**language**	**#pkg**	**#rel**	**#dep**
Cargo	2014	Rust	35K	183K	796K
npm	2010	JavaScript	1,218K	9,383K	48,695K
Packagist	2012	PHP	180K	1.520K	4,727K
RubyGems	2004	Ruby	155K	956K	2,396K
			1,588K	12,041K	56,615K

Table 1 summarises the curated dataset, reporting the number of packages

(#pkg), package releases (#rel), and dependencies (#dep) that are considered for the empirical analysis. The data and code to replicate the analysis are available on `https://doi.org/10.5281/zenodo.4013419`.

4. Research Questions

4.1. How Prevalent Are 0.y.z Packages?

Since the results of our analysis are only relevant if a sufficient number of packages in each distribution are still in their initial development phase, we compute for each distribution, on a monthly basis, the proportion of packages whose latest available release is 0.y.z. Figure 1 shows the evolution of this proportion relative to the number of packages distributed at that time.

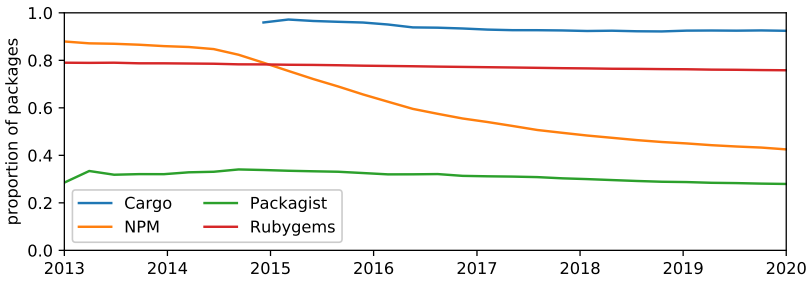

Figure 1: Evolution of the proportion of packages whose latest release is 0.y.z.

We observe a very high proportion of initial development packages for Cargo (94% on average). At the last observation time (December 2019), 92.4% of its packages were still 0.y.z packages. For RubyGems, we observe a quite stable proportion of 0.y.z packages over time, of 77.4%. On the other side of the spectrum we find Packagist, with 31% of 0.y.z packages on average, and 30% at the last observation time. npm is the only one of the considered distributions to exhibit a non stable proportion of 0.y.z packages. Indeed, we observe from April 2014 onwards that the proportion of 0.y.z packages went from 85.5% to 42.5%. This is a consequence of npm policies aiming to reduce the use of 0.y.z releases, notably by changing the initial version of packages created through `npm init` to 1.0.0 instead of 0.1.0 "*since the semver spec is weirdly magical about 0.x.y versions, and we cannot ever hope to get everyone to believe what the correct interpretation of 0.x versions are.*"[5]

It could be the case that the high proportions of 0.y.z packages we observed are due to *"many open source projects [that] languish in the 0.x.x state because the developer lost interest and stopped working on it"* [1], i.e., due to packages no longer being maintained and thus preventing them from ever reaching a

[5]See `https://github.com/npm/init-package-json/commit/363a17bc3`

≥1.0.0 release. To check whether inactive packages could have influenced the observations we derived from Figure 1, we repeated this analysis by removing all packages that were not active during the last year (i.e., packages that have not released a new version during the last 12 months). Figure 2 shows the evolution of the proportion of *active* 0.y.z packages (straight lines) relative to the number of active packages. To ease the comparison, we also report the proportion of *all* (i.e., both active and inactive) 0.y.z packages (dotted lines) as in Figure 1. [6]

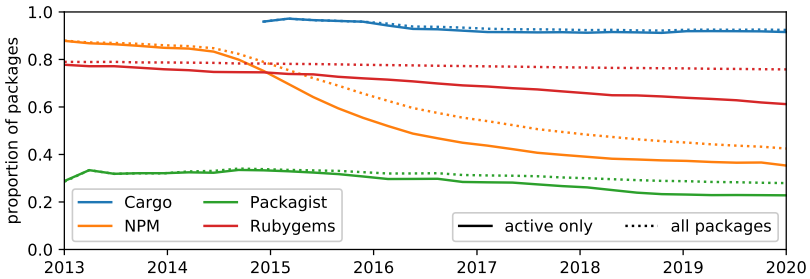

Figure 2: Evolution of the proportion of active 0.y.z packages (straight lines).

We observe that the proportions of *active* 0.y.z packages (straight lines) follow closely the proportions of *all* 0.y.z packages (dotted lines). This is especially visible in Cargo where the difference is only of 0.8% (= 92.4 − 91.6) in the last considered snapshot. We also observe that the difference remains limited for for Packagist (5.2% = 28 − 22.8) and for npm (7.2% = 42.5 − 35.3), indicating that the proportion of 0.y.z packages does not depend on whether they are active or not. Even if the difference is more pronounced in RubyGems (14.6% = 75.8−61.2), there are still proportionally many more 0.y.z than ≥1.0.0 packages amongst the active ones. We can therefore reject our hypothesis that the high proportions of 0.y.z packages we observed in Figure 1 are due to the presence of many 0.y.z packages being no longer maintained.

To confirm that 0.y.z packages represent a large part of the activity in the considered distributions, we computed the monthly proportion of 0.y.z releases relatively to the total number of new releases. Figure 3 shows the evolution of these proportions. We observe that 0.y.z packages are responsible for the large majority of package releases in Cargo (median 90.8%), RubyGems (median 74.3%) and npm (median 58.8%). On the other hand, "only" one out of four releases in Packagist is due to 0.y.z packages (median 27.1%). We also observe that these proportions are slightly decreasing over time in all package distributions. This is especially visible for npm from April 2014 onwards, a consequence of the new npm policies about initial development releases, as mentioned above. However, at the end of the observation period, 0.y.z packages still account for

[6]The proportion of ≥1.0.0 packages can be obtained by taking the complement of the proportion of 0.y.z packages.

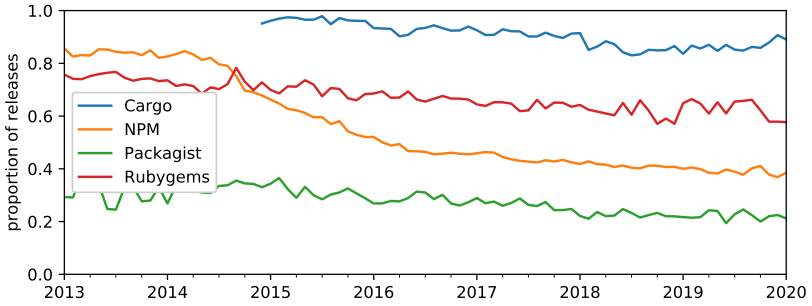

Figure 3: Evolution of the monthly proportion of 0.y.z releases.

89% of all package releases in Cargo, for 57.7% in RubyGems, for 38.6% in npm and for 21.2% in Packagist. These proportions are quite similar to the proportions of active 0.y.z packages in each distribution, indicating that 0.y.z and ≥1.0.0 packages do not really differ in term of release activity.

> The considered package distributions contain many 0.y.z packages: more than one out of five in Packagist, more than one out of three in npm, more than three out of five in RubyGems, and more than nine out of ten in Cargo. 0.y.z packages are nearly as active as ≥1.0.0 packages, and are responsible for the majority of all package releases in Cargo and RubyGems.
> The release policies of Cargo and RubyGems should be adapted to incite package maintainers to move out of the zero version space, the same way as npm has successfully done in 2014.

4.2. Do Packages Get Stuck in the Zero Version Space?

If one assumes that 0.y.z packages are still under initial development, then they are eventually expected to reach a ≥1.0.0 release reflecting their maturation. However, developers are *"hesitant to increment their projects to 1.0.0 and stay in 0.x.x for a very long time, and possibly forever."* [1]

To measure how long it takes to reach a ≥1.0.0 release in each of the considered package distribution, we rely on the statistical technique of survival analysis (a.k.a. event history analysis) [18] to model the time for the event "package reaches ≥1.0.0" to occur as a function of the time elapsed since the first release of that package. Survival analysis estimates the survival rate of a given population of subjects (packages in our case), i.e., the expected time duration until the event of interest (from the first available release until the first ≥1.0.0 release) occurs. Survival analysis takes into account the fact that some observed subjects may be "censored", either because the event was observed

prior to the observation period (i.e., the first considered release of the package was already ≥1.0.0), or not observed during the observation period (i.e., the package never reached a ≥1.0.0 release). A common non-parametric statistic used to estimate survival function is the Kaplan-Meier estimator [19].

Survival functions define the probability of surviving past time t or, equivalently, the probability that the event has not occurred yet at time t (i.e., a package in the distribution has not reached a ≥1.0.0 release). Figure 4 shows the Kaplan-Meier survival functions for the four considered distributions. Based on this probability, the complement probability that a package reaches ≥1.0.0 within a given time can be easily computed.

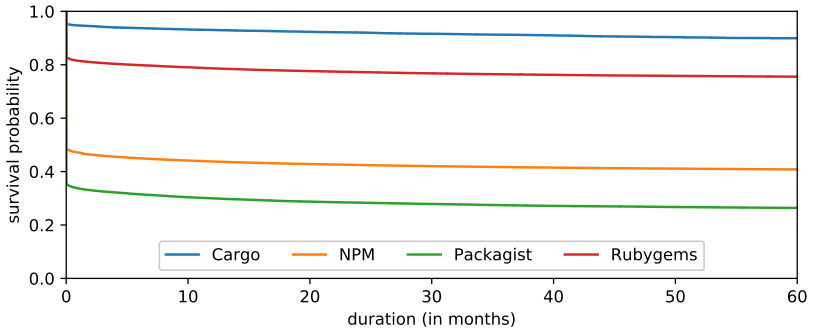

Figure 4: Kaplan-Meier survival curves for the probability that a "package reaches ≥1.0.0" as a function of the time elapsed since the first release of that package.

We observe that the survival probability over time is mainly driven by the survival probability at time 0, i.e., by the proportion of packages being directly released with a ≥1.0.0 release. While the probability to remain 0.y.z slightly decrease over time, this decrease is rather limited (from 0.06 for Cargo to 0.10 for Packagist). On average, the survival probability decreases by less than 0.02 per year, a direct consequence of many packages having not reached yet the 1.0.0 barrier.

To confirm that most packages have not reached the 1.0.0 barrier yet, we computed the proportion of packages whose first distributed release was already mature (≥1.0.0), packages that eventually crossed the 1.0.0 barrier, and packages remaining in their 0.y.z phase. Figure 5 shows the proportion of such packages for each package distribution.

We observe that most packages in Cargo (92.4%) and in RubyGems (75.8%) only have 0.y.z releases. On the other hand, the majority of Packagist packages (62.6%) only have ≥1.0.0 releases. For npm, there is a more or less equal proportion of packages having only 0.y.z releases and packages having only ≥1.0.0 releases. We also observe that, regardless of the package distribution, less than one out of ten packages went from a 0.y.z to a ≥1.0.0 release. This represents 1.176 packages in Cargo (3.4%), around 103K in npm (8.4%), 17K packages in Packagist (9.5%) and 13K packages in RubyGems (8.3%).

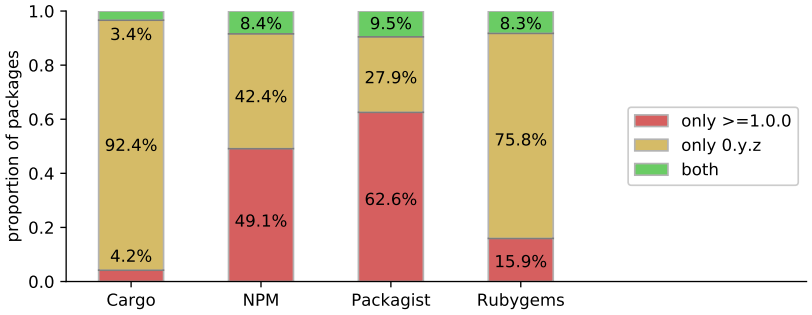

Figure 5: Proportion of packages having only ≥1.0.0 releases, only 0.y.z releases, or both.

Focusing exclusively on packages that traversed the 1.0.0 barrier, we computed the duration between their first 0.y.z release and their first ≥1.0.0 release. Figure 6 presents the cumulative proportion of packages having reached the 1.0.0 barrier in function of the duration in time (left) and in terms of number of intermediate releases (right).

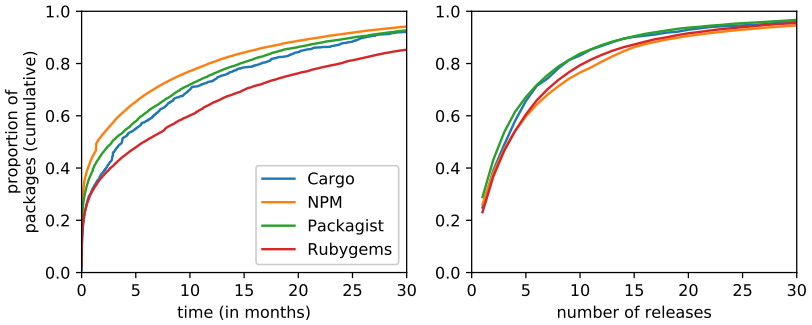

Figure 6: Duration (in months) and number of releases between first 0.y.z and first ≥1.0.0 release.

We observe that a majority of packages take only a few months and a few release updates to reach ≥1.0.0. The median duration varies between 1.4 months (for npm) and 5.8 months (for RubyGems) while the median number of release updates is 3 for Packagist and 4 for the other package distributions. Yet there are many packages that took much longer to reach ≥1.0.0. Over one out of five packages (35.3% for RubyGems, 26.7% for Cargo, 24.3% for Packagist and 19.6% for npm) needed more than a year to reach ≥1.0.0, and many packages more than 2 years (19.9% in RubyGems, 12.3% in Cargo, 10.8% in Packagist and 8.8% in npm). There are even 6.2% of all packages in RubyGems that needed more than 4 years to reach ≥1.0.0. For comparison, they are less than 1.8% in the other distributions.

> Many packages get stuck in the zero version space. The probability to reach ≥1.0.0 increases by less than 0.02 per year. Less than 10% of all packages went from 0.y.z to ≥1.0.0 releases. While a majority of them only took a few months and a few updates to reach ≥1.0.0, one out of five of them took more than one year to cross the 1.0.0 barrier, and many of them took even more than two years, especially in RubyGems. Package maintainers should not be afraid of crossing the 1.0.0 barrier. Packages that have been developed for years and that are indubitably ready for production should receive a major version 1 or higher.

4.3. Are 0.y.z Releases Published More Frequently than ≥1.0.0 releases?

One would expect packages under initial development to publish new releases more frequently than mature packages. This is notably assumed by the semver policy that states that "*major version zero is all about rapid development*". To verify this assumption we computed the distribution of the average time per package between consecutive releases (a.k.a. the period, corresponding to the reciprocal of the release frequency), for 0.y.z and ≥1.0.0 releases respectively. Figure 7 shows the boxen plots [20] for these distributions.

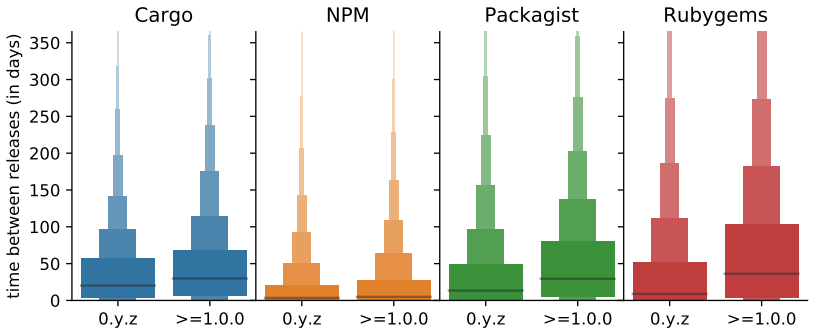

Figure 7: Distributions of the average number of days between consecutive releases, for 0.y.z and ≥1.0.0 releases.

We observe that for all studied package distributions, 0.y.z releases are more frequently published than ≥1.0.0 releases (i.e., the average time between releases is higher in ≥1.0.0 than in 0.y.z releases). For instance, for Cargo, the median was 20.2 days for 0.y.z and 29.9 days for ≥1.0.0 releases. For npm the median values were 3.4 and 4.9 days respectively, for Packagist 13.4 and 29.4 days, and for RubyGems 8.9 and 36.2 days. To confirm the statistical significance of these differences between the release frequencies of 0.y.z and ≥1.0.0 releases, we carried out one-sided Mann-Whitney-U tests [21]. The null hypothesis, stating that there is no difference between the release frequencies of 0.y.z and ≥1.0.0 releases, was rejected for all four package distributions with $p < 0.01$ (adjusted

after Bonferroni-Holm method to control family-wise error rate [22]). However, the effect size (measured using Cliff's delta d [23]) revealed that the observed differences were *negligible* for Cargo ($|d| = 0.099$) and npm ($|d| = 0.041$), and *small* for Packagist ($|d| = 0.179$) and RubyGems ($|d| = 0.240$), following the interpretation of $|d|$ by Romano et al. [24].

We also observe that the time to release a new version is lower in npm than in the other package distributions, i.e., npm packages seem to publish releases more frequently. Mann-Whitney-U tests confirmed this observation with statistically significant differences ($p < 0.01$), implying that releases in npm are indeed published more frequently than releases in Cargo, Packagist and RubyGems. However, the effect size turned out to be *small* in all cases: $|d| = 0.327$ for Cargo, $|d| = 0.325$ for Packagist and $|d| = 0.185$ for RubyGems. We also compared the release frequency of packages in Cargo, Packagist and RubyGems. While we found statistically significant differences between them, the effect size was always *negligible* ($0.028 \leq |d| \leq 0.108$).

The previous analysis compared the release frequency between all 0.y.z releases and all ≥1.0.0 releases of a *package distribution*, providing insights for the whole package distribution at once. The following analysis focuses on the evolution of the release frequency between the 0.y.z and ≥1.0.0 releases of *individual packages*. For each package, we compared the frequency of its 0.y.z releases to the frequency of its ≥1.0.0 releases.[7]

The boxplots in Figure 8 show the distribution of the ratio between the release frequency of ≥1.0.0 and 0.y.z releases of each package. A ratio above 1 implies that the ≥1.0.0 releases of a package are more frequently published than its 0.y.z releases, while a ratio below 1 means that the 0.y.z releases of the package are more frequently published. For obvious reasons, only packages having both 0.y.z and ≥1.0.0 releases were considered for this analysis.

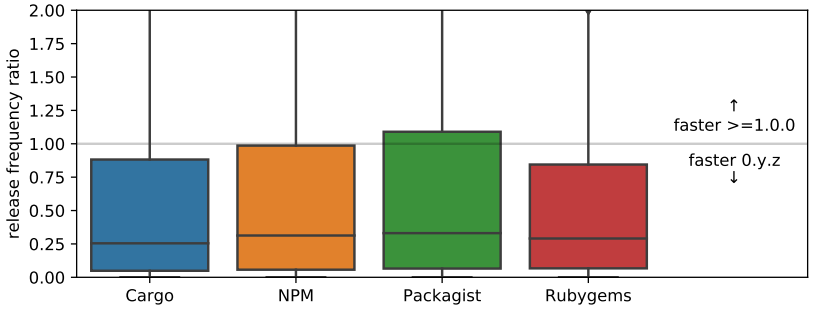

Figure 8: Distributions of the ratio between the release frequencies of ≥1.0.0 and 0.y.z releases for each package.

[7]The release frequency of 0.y.z (resp. ≥1.0.0) releases is obtained by dividing the number of 0.y.z (resp. ≥1.0.0) releases by the time between the first and last 0.y.z (resp. ≥1.0.0) releases.

We observe that in all considered package distributions, the large majority of packages have a ratio below 1, indicating that 0.y.z releases are published more frequently than ≥1.0.0 releases. The median ratio oscillates between 0.25 (for Cargo) and 0.33 (for Packagist), indicating that 0.y.z versions are released 3 to 4 times more frequently than ≥1.0.0 releases. There are only from 22% (for RubyGems) to 26.7% (for Packagist) packages whose 0.y.z release frequency is lower or equal to their ≥1.0.0 release frequency.

> 0.y.z releases are published more frequently than ≥1.0.0 releases, but the effect is small for Packagist and RubyGems, and negligible for Cargo and npm. The release frequency of most packages is higher for their 0.y.z releases than for their ≥1.0.0 releases. Regardless of their version number, package releases are published more frequently in npm than in the other distributions.

4.4. Are 0.y.z Package Releases Required by Other Packages?

If one assumes that 0.y.z packages are still under initial development, it could be considered unsafe to rely on them since they are expected to be less complete and less stable than production-ready packages. This is confirmed by the semver policy [4]: "*If your software is being used in production, it should probably already be 1.0.0.*" Moreover, such 0.y.z packages are likely to require extra effort from maintainers of packages depending on them. Indeed, since "*anything may change at any time [and] the public API should not be considered stable*", dependent packages are more likely to face breaking changes with 0.y.z packages than with ≥1.0.0 packages. semver even recommends that "*If you have a stable API on which users have come to depend, you should be 1.0.0*".

This research question therefore studies the extent to which packages rely on such 0.y.z packages. Since the dependencies of a package can evolve over time, we consider the dependencies expressed in the latest available release of each package, hence reflecting the state of the latest snapshot in the package distributions. Table 2 reports the proportion of dependent packages (% sources) relying on at least one 0.y.z package, and the proportion of required packages (% targets) being used by at least one ≥1.0.0 package. We distinguish between 0.y.z and ≥1.0.0 sources and targets.

The reported proportions vary greatly from one package distribution to another. For instance, a large majority ($88.8\% = 82 + 6.8$) of the dependent packages in Cargo rely on at least one 0.y.z release. 82% of these dependent packages are 0.y.z while only 6.8% are ≥1.0.0. For Packagist and RubyGems, the inverse is true: a large majority of dependent packages ($86.6\% = 20.3 + 66.3$ for Packagist, and $77.6\% = 56.7 + 20.9$ for RubyGems) rely exclusively on ≥1.0.0 package releases. npm falls somewhere in the middle of both extremes, with 43.5% ($= 23.7 + 19.8$) dependent packages relying on at least one 0.y.z package release, and the remaining 56.5% ($= 21.3 + 35.2$) relying exclusively on ≥1.0.0

Table 2: Proportion of source and target packages, depending on or required by 0.y.z and ≥1.0.0 packages.

package distribution	source	target 0.y.z	target ≥1.0.0	target 0.y.z	target ≥1.0.0
Cargo	**0.y.z**	82.0	9.9	76.9	5.9
	≥1.0.0	6.8	1.4	11.5	5.8
npm	**0.y.z**	23.7	21.3	23.9	10.4
	≥1.0.0	19.8	35.2	14.6	51.1
Packagist	**0.y.z**	7.9	20.3	13.1	10.4
	≥1.0.0	5.5	66.3	6.1	70.4
RubyGems	**0.y.z**	17.4	56.7	20.9	34.7
	≥1.0.0	5.0	20.9	8.6	35.8
proportionally to		**% sources**		**% targets**	

package releases. Nevertheless, in all four package distributions there is still a large number of dependent packages relying on at least one 0.y.z package release.

When considering these numbers proportionally to the set of required packages (i.e., % targets), we observe that 88.4% (= 76.9 + 11.5) of the required packages in Cargo are 0.y.z packages. This is in stark contrast with Packagist where this proportion is only 19.2% (= 13.1 + 6.1). npm and RubyGems fall somewhere in between, with 38.5% (= 23.9 + 14.6) of the required npm packages being 0.y.z packages, and 29.5% (= 20.9 + 8.6) for RubyGems. This indicates that in all four package distributions, at varying degrees, many 0.y.z packages are still being used by other packages, including ≥1.0.0 ones. This is rather counter-intuitive: even though common wisdom says that 0.y.z packages are more likely to be unstable because they are still under inital development, package maintainers frequently depend on them.

To determine to which extent 0.y.z and ≥1.0.0 packages are required, we computed for each package distribution their number of dependent packages (i.e., reverse dependencies). If a package has both 0.y.z and ≥1.0.0 releases being required, we distinguish between packages depending on its 0.y.z releases from packages depending on its ≥1.0.0 releases. We do so by looking at whether the highest release accepted by the dependency constraint is a 0.y.z or a ≥1.0.0 release, complying with the default behaviour of the package managers used in the considered package distributions. Figure 9 shows the boxen plots of the distribution of the number of dependent packages for required packages, distinguishing between packages depending on their 0.y.z or on their ≥1.0.0 releases, respectively.

Regardless of the considered package distribution, we only observe small differences between the number of dependent packages for required 0.y.z and ≥1.0.0 packages. On average, the number of dependent packages for 0.y.z packages oscillates between 2.1 (for Packagist) and 4.6 (for Cargo), while for ≥1.0.0 packages it is comprised between 6.9 (for Packagist) and 10.7 (for Cargo). In all cases, the median number of dependent packages is 1. We statistically compared

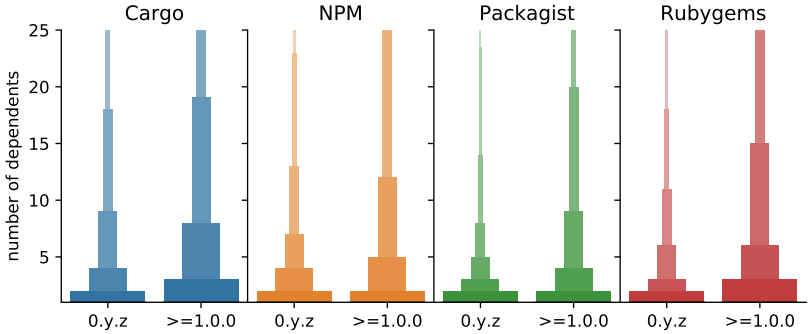

Figure 9: Distributions of the number of dependent packages for required 0.y.z and ≥1.0.0 packages.

for each package distribution the number of dependent packages for 0.y.z and ≥1.0.0 packages using a Mann-Whitney-U test to find evidence of a statistical difference. The null hypothesis was rejected for all four package distributions ($p < 0.01$ after Bonferroni-Holm correction), indicating that ≥1.0.0 packages are required more often. However, the effect size was *negligible* for all four package distributions ($0.066 \leq |d| \leq 0.108$).

> Many packages are depending on 0.y.z packages, ranging from 13.4% of all dependent packages in Packagist to 88.8% in Cargo. Many 0.y.z packages are required by other packages, ranging from 19.2% of all required packages in Packagist to 88.4% in Cargo. There is no practical difference between the number of packages depending on 0.y.z and ≥1.0.0 packages.
> Maintainers of 0.y.z packages should strive to make their packages cross the 1.0.0 barrier if they are used by other production-ready packages.

4.5. How Permissive Are Dependency Constraints Towards Required 0.y.z Packages?

This research question focuses on a variation on the theme of unstable initial development packages. Under the premise that 0.y.z packages are unstable, the semver policy assumes that any update of such a package could introduce backward incompatible changes: "*Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable.*" [4]

When specifying package dependencies, package maintainers make use of dependency constraints to specify which releases of the required package are allowed to be installed. To ease the definition of such constraints in combination with semver, package distributions provide specific notations to accept

patches (e.g., ∼1.2.3) or minor releases (e.g., ^1.2.3). However, these notations are not the only way to specify dependency constraints (e.g., >=1.2.3, <2.0.0 is equivalent to ^1.2.3), and their interpretation is not always consistent across package distributions.[8] To take these differences into account, we wrote a dependency constraint parser[9] to convert the constraints using specific notations provided by each package distribution into a generic version range notation based on intervals [25]. For example, ∼1.2 is converted to the right-open interval $[1.2.0, 1.3.0)$ for Cargo and npm, and to the right-open interval $[1.2.0, 2.0.0)$ in Packagist. Based on this version range notation, we can easily identify whether a dependency constraint allows new patches, new minor and/or new major releases of the required package to be automatically installed. For example, range $[1.2.0, 1.3.0)$ accepts new patches because it contains version $1.2.x$ for $x > 0$, and does not accept new minor releases because it does not contain version $1.x.0$ for $x > 2$.

For each considered package distribution we computed the monthly proportion of dependency constraints targeting 0.y.z releases that accept at least patches or minor releases. These proportions are shown in Figure 10 relative to the total number of constraints targeting 0.y.z releases defined in newly distributed releases for each month. To avoid the analysis being biased by packages having many releases during the month, only the latest available release of each package was considered for each month.

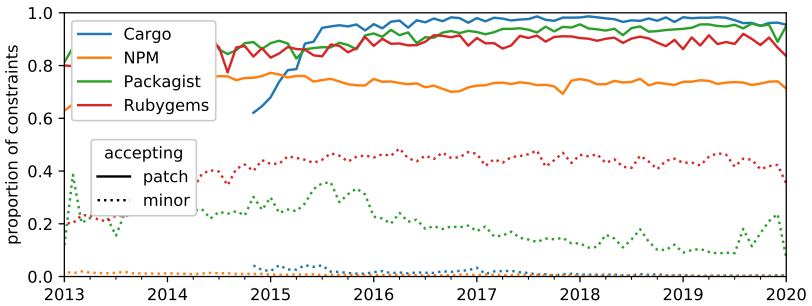

Figure 10: Evolution of the monthly proportion of dependency constraints accepting at least patches or minor 0.y.z releases.

We observe that in all four distributions a large proportion of dependency constraints is accepting at least patches, from around 73% for npm to 94.2% for Cargo. This proportion remains mostly stable over time. The proportion of dependency constraints accepting minor releases as well depends on the considered package distribution: it is close to zero for Cargo and npm while, on the other hand, it fluctuates around 40% in RubyGems. Packagist sits in the middle of these extremes, with a steadily decreasing proportion of constraints accepting

[8] The reader is invited to consult Table 2 of [6] for more details.
[9] See `https://doi.org/10.5281/zenodo.4013419`

minor releases since mid-2015, from 33.1% to around 10% in 2019.

Focusing on the latest snapshot of each package distribution, Figure 11 reports the proportion of dependency constraints accepting at most patches, minor or major releases, separating between dependencies targeting 0.y.z releases and ≥1.0.0 releases, respectively.

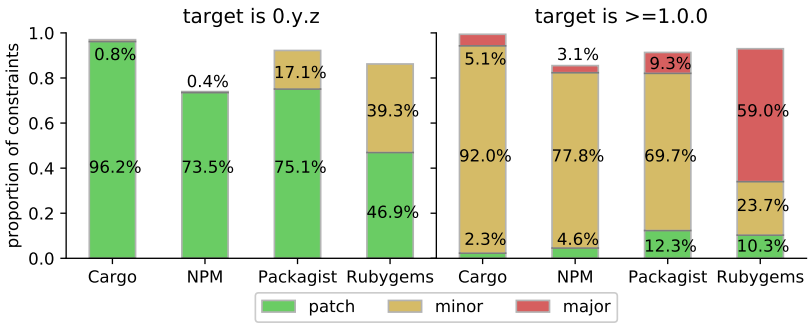

Figure 11: Proportion of dependency constraints accepting at most patches, minor or major releases, grouped by target releases.

In order to be semver-compliant, only strict constraints (i.e., constraints that only accept a single version) to 0.y.z package releases should be allowed to avoid the risk of introducing breaking changes. However, we observe that the large majority of the dependencies targeting 0.y.z releases are more permissive: they allow patches to be automatically installed, from 73.9% (=73.5 + 0.4) in npm to 97% (=96.2 + 0.8) in Cargo. Packagist and RubyGems are even more permissive, since a non-negligible proportion of dependency constraints to 0.y.z package releases accept minor release updates as well (17.1% of all constraints targeting 0.y.z releases in Packagist, and 39.3% in RubyGems).

For comparison, we carried out the same analysis for dependencies targeting ≥1.0.0 releases. For those cases, the semver policy considers it safe to accept minor release updates (since minor releases are expected to contain only backward compatible changes). We indeed observe that the large majority of dependencies towards ≥1.0.0 releases accept minor releases as well. For instance, we found 92.5% (= 92 + 0.5) of such dependencies in Cargo, 78.1% (= 77.8 + 0.3) in npm, and 78.9% (= 69.7 + 9.2) in Packagist. For RubyGems, we found 82.7% (= 23.7 + 59) of such dependencies, a consequence of the presence of 59% of dependencies accepting major releases as well. For RubyGems, the high proportion of dependencies accepting minor 0.y.z releases and major ≥1.0.0 releases indicates that its packages are not semver-compliant, not even for ≥1.0.0 releases [6].

Most dependencies towards 0.y.z releases accept new patches, indicating that these patches are expected to be backwards compatible. As such, the considered package distributions adopt a policy that is more permissive than semver for 0.y.z releases. Packagist and RubyGems are even more permissive, since more than one dependency constraint out of six also accepts minor releases.
This relaxation w.r.t. semver should be made explicit, or the semver policy should be loosened to allow package maintainers to specify backwards compatible updates for 0.y.z releases.

4.6. Do GitHub Repositories for 0.y.z Packages Have Different Characteristics?

So far, we have not really been able to discern any difference between 0.y.z and ≥1.0.0 packages. If we assume that ≥1.0.0 packages are more production-ready, stable and mature than 0.y.z packages, we can expect their git repositories to have more stars, more forks, more contributors or less open issues than the git repositories of 0.y.z packages. In this research question, we aim to compare 0.y.z and ≥1.0.0 packages based on the characteristics of their git repository, focusing on the ones being hosted on GitHub, the most popular distributed collaborative development platform.

Since not all packages have an associated repository on GitHub, Table 3 reports only the proportion of 0.y.z and ≥1.0.0 packages that have a known GitHub repository in our dataset.

Table 3: Proportion of packages with a known GitHub repository.

distribution	**0.y.z packages**	**≥1.0.0 packages**	**all packages**
Cargo	72.1%	85.9%	73.1%
npm	62.5%	59.8%	61.0%
Packagist	94.8%	94.3%	94.4%
RubyGems	65.3%	70.9%	66.7%

We observe that a large majority of all packages have an associated repository on GitHub, from 61% for npm to 94.4% for Packagist. This higher proportion for Packagist is a consequence of the way packages are made available. Indeed, to submit a new package on Packagist, one has to provide the URL of a public repository. We also observe a slight difference between the proportion of 0.y.z and ≥1.0.0 packages having a repository on GitHub: there are proportionally more repositories for ≥1.0.0 packages in Cargo (85.9% versus 72.1%) and in RubyGems (70.9% versus 65.3%) while there are proportionally more repositories for 0.y.z packages in npm (62.5% versus 59.8%) and in Packagist (94.8% versus 94.3%).

For the GitHub repository associated with each package, we used our dataset to extract the number of stars, forks, contributors, open issues and the size of

the git repository (expressed in megabytes). We also extracted the number of dependent repositories, i.e., software projects developed in GitHub repositories and that depend on the given package. Figure 12 shows the boxen plots of the distributions of these characteristics for each package distribution, distinguishing between repositories hosting 0.y.z and ≥1.0.0 packages.

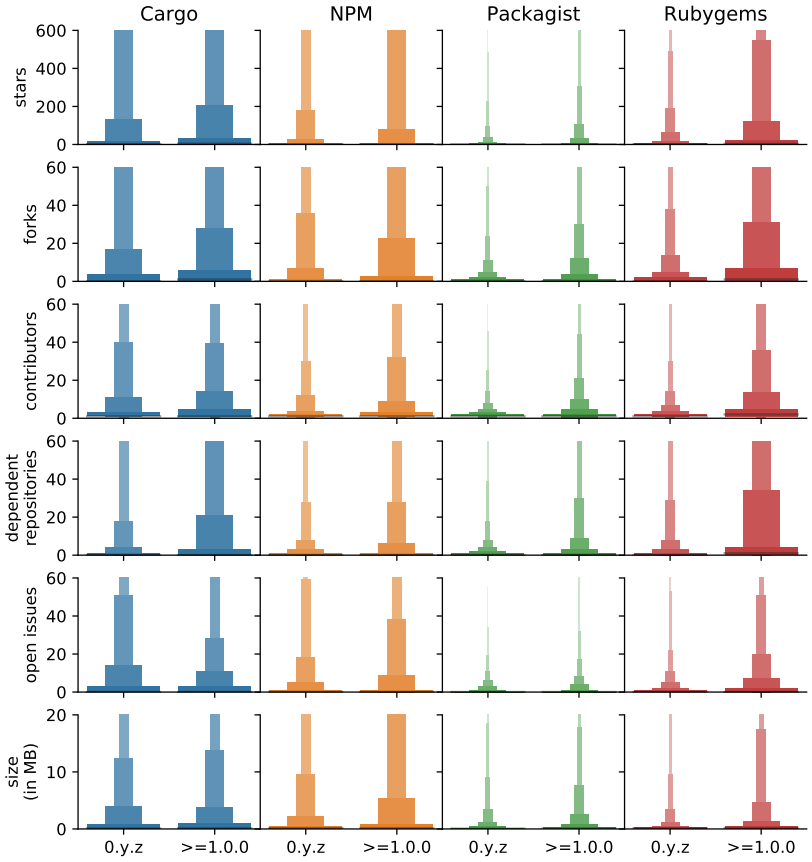

Figure 12: Distributions of the number of stars, forks, contributors, dependent repositories, open issues and git repository size (in MB) for 0.y.z and ≥1.0.0 packages.

We observe a slight difference between 0.y.z and ≥1.0.0 repositories for all characteristics and for all four package distributions. The differences are especially visible in RubyGems. We carried out Mann-Whitney-U tests between 0.y.z and ≥1.0.0 package repositories for each characteristic to confirm these differences. The null hypothesis stating that there is no difference between 0.y.z and ≥1.0.0 package repositories was consistently rejected in all cases ($p < 0.01$ after Bonferroni-Holm correction), with the notable exception of open issues in Cargo. This confirms that ≥1.0.0 package repositories have more stars, forks, contributors, dependent projects, and open issues (except in Cargo). More-

over, they are larger than 0.y.z package repositories. Nevertheless, the effect size turned out to be *negligible* for all comparisons in Cargo, npm and Packagist ($0.006 \leq |d| \leq 0.132$) and *small* in RubyGems ($0.156 \leq |d| \leq 0.258$).

> A large majority of 0.y.z and ≥1.0.0 packages have an associated repository on GitHub. Repositories for ≥1.0.0 packages have slightly more stars, forks, contributors, dependent projects and open issues (except in Cargo), but the differences are negligible for Cargo, npm and Packagist, and small for RubyGems.

5. Discussion

This section aims to discuss about the lessons learned from the empirical analysis, as well as the recommendations that could be made to package maintainers and managers of package distributions based on these findings.

5.1. Qualitative evidence

The results reported in this paper were based on quantitative evidence relying on historical analysis of package management data. To complement this quantitative analysis, we conducted a single question poll on Twitter and LinkedIn in December 2020. The poll aimed to obtain insights in whether practitioners effectively consider it risky to depend on packages that are still in version 0.y.z. The single multiple-choice question was:

> *"As a software developer, you need to depend on an open source package distributed through some package manager (like npm, Maven, Cargo, Packagist, RubyGems). Would you trust depending on a package with major version 0 (e.g., version 0.5.1)?"*

Table 4: Number of answers for each of the 4 possible responses to the multiple-choice question.

response	**# answers**
No	7
Only if there is no alternative	21
Only after checking	45
Sure	37
Total	110

We received 110 responses in total, of which 58 on LinkedIn and 52 on Twitter. The results are summarised in Table 4. From these results we conclude that the large majority of respondents considers it risky to depend on 0.y.z package releases. This corresponds to the common convention that 0.y.z packages are assumed to be under initial development and therefore potentially less stable than ≥1.0.0 releases.

Lessons learned. Common belief suggests that 0.y.z package releases are less stable than ≥1.0.0 ones. 66.4% of respondents perceive depending on 0.y.z package releases as risky: they would prefer not to depend on 0.y.z packages, or only if there is no alternative or after doing additional checks on that package.

5.2. The 1.0.0 barrier

The results we obtained from the quantitative analysis in this paper seem to be in shrill contrast with the common wisdom that was confirmed by the qualitative analysis presented in Section 5.1. We observed little difference between 0.y.z and ≥1.0.0 packages, implying that many 0.y.z packages can be assumed to be production ready and safe to use. Given this little difference, a kind of psychological barrier related to version 1.0.0 could perhaps explain why so few packages reach a ≥1.0.0 release. A major version 1 is usually associated with the promise of a stable API and a mature library. For npm, a developer witnesses that *"[he] tends to associate 1.0.0 with a finished project, including tests, documentation, a nice landing page, and a lot of sample code."*. However, he agrees that *"while all of that is certainly important in its own right, these extra components of a project actually have nothing to do with the version number"* [1].

We believe that most developers of packages in the 0 version space avoid to cross the 1.0.0 barrier in order to keep the freedom to make API (breaking) changes, and to not have to commit to the (overly optimistic and unrealistic) bug-free nature of ≥1.0.0 releases, even if their package already reached this degree of maturity. Because of this, developers have to rely on other ways to assess the maturity of 0.y.z packages, as witnessed by another Rust developer who *"[has] to go read the code to see if this is a '0.1.0' package which is basically finished, or a '0.1.0' full of partially implemented functionality"* [2].

It is not unusual that the decision to cross the 1.0.0 barrier is based on other factors, such as providing more functionalities or polishing the package. For example, Tom Augspurger from the pandas developer team testifies that *"pandas has been 'production ready' for a while now, in the sense that it's used in production at many institutions. But we still had a few major items we wanted to iron out before calling 1.0"*.[10] Similarly, Piotr Solnica, maintainer of dry-validation, one of the most used packages in RubyGems, explains *"[they] have got a long backlog in the issue tracker and [he] would like to address all of those issues ASAP and make the codebase simpler AND add features that are even more powerful than what we have already. The final goal is to turn this into 1.0.0 in a couple months from now."*.[11]

[10] `https://jaxenter.com/python-pandas-1-0-0-tom-augspurger-167593.html`
[11] `https://discourse.dry-rb.org/t/plans-for-dry-validation-dry-schema/215`

There are plenty examples of popular 0.y.z packages being developed for years, known for their stability and maturity and being used in production by thousands of users. One such example is style-loader, one of the most famous libraries on npm. Despite its widespread use (it has more than 10K dependent packages, and several millions weekly downloads), it only reached its first $\geq$1.0.0 release in August 2019, after more than 7 years of development and 54 releases. Another example is the syn package in Cargo. Its first $\geq$1.0.0 release was reached after 3 years of development and 122 releases. At that time, it was already used by one third of all packages in Cargo. For its developers, the decision to release a 1.0.0 version *"signifies that Syn is a polished and stable library and that it offers a user experience we can stand behind as the way we recommend for all Rustaceans to write procedural macros"*.[12]

It is not surprising that we did not observe any fundamental difference between 0.y.z packages and $\geq$1.0.0 packages, both in terms of update frequency and usage by other packages, even if the common belief suggests such a difference holds. This belief is reinforced by how package distributions and versioning policies treat both types of packages. Indeed, we found that npm, Packagist and Cargo make an explicit distinction between how dependency constraints are treated for 0.y.z and $\geq$1.0.0 releases, based on the presumed degree of maturity.

An alternative approach consists of not associating a different versioning policy to 0.y.z and $\geq$1.0.0 releases. This is the case for Haskell packages, for which the official versioning policy explicitly states that "*packages with a zero major version provide the same contractual guarantees as versions released with a non-zero major version*".[13] This does not seem to be a perfect solution either, since in practice it does not encourage maintainers to cross the 1.0.0 barrier either: "*an easily spottable plague of an absolute majority of Haskell packages is that they get stuck in the 0.x.x version space, thus forever retaining that 'beta' feeling even if the package's API remains stable for years and has dependencies counted by thousands*".[14]

[12] `https://github.com/dtolnay/syn/issues/687`
[13] `https://pvp.haskell.org/faq/`
[14] `https://www.reddit.com/r/haskell/comments/31e3jj/`

Lessons learned. Most packages remain stuck in the zero version space. There seems to be a psychological barrier associated to version 1.0.0. The use of different rules and versioning policies for 0.y.z and ≥1.0.0 releases by the considered package distributions only seems to reinforce this barrier.

Recommendations. There is no fundamental reason why 0.y.z releases should not fulfil the same contracts and promises as ≥1.0.0 releases, especially as soon as a package is being distributed and used by others. Package maintainers should strive to cross the 1.0.0 barrier, especially if their packages have been developed for years and are indubitably ready for production.

5.3. Maturity of distributed packages

Our analysis found little difference between 0.y.z and ≥1.0.0 packages, implying that many 0.y.z packages can be assumed to be production-ready and safe to use. This could be explained by the fact that, as mentioned by a Rust developer, *"publishing already implies some degree of production readiness"* [2]. Although it is probably true that packages being available in package distributions are at least functional and offer a minimum of features, none of the four consider package distributions actually require their distributed packages to achieve some degree of production readiness.

We have not found any guidelines in the documentation of these package distributions specifying or suggesting the minimal requirements a package must or should fulfil for its distribution. At most, the documentation of RubyGems mentions that *"testing your gem is extremely important"* and that *"developers tend to view a solid test suite (or lack thereof) as one of the main reasons for trusting that piece of code."*[15]

We are convinced that package distributions and package developers would greatly benefit from guidelines defining the minimum requirements for a package to be distributed. For example, CRAN, the official package distribution for the R programming language, states that *"CRAN hosts packages of publication quality and is not a development platform. A package's contribution has to be non-trivial."* It imposes several requirements with the respect to the quality and maturity a package must have in order to be accepted for distribution.[16]

The lack of guidelines in the considered package distributions means that anyone can basically publish anything in these package distributions, including immature, unstable, incomplete or even non-working packages. This explains why we found packages with clearly deviating and undesirable behaviour in our dataset, as explained in Section 3.

[15] `https://guides.rubygems.org/make-your-own-gem/`

[16] See `https://cran.r-project.org/web/packages/policies.html` for more details.

Imposing requirements and/or responsibilities on the packages that can be distributed in a package distribution could prevent many packages from being "officially distributed". For example, a maintainer may not be willing or able to meet these requirements, or may not be willing to commit to certain responsibilities, especially if they are not limited in time. However, just because a package cannot or is not distributed on a package distribution does not prevent other packages or users from making use of it. Indeed, the package manager tools of the considered package distributions all provide an easy way to install or depend on packages from another source (e.g., from a private package distribution, from a git repository, etc.).

> **Lessons learned.** None of the considered package distributions provide guidelines on the minimal requirements that a package should fulfil to be distributed.
>
> **Recommendations.** Package distributions should clarify the expectations and responsibilities associated with distributing a package through the package manager.

5.4. Version policies of package distributions

We found in Section 4.1 that npm is the only one of the considered distributions that exhibits a clearly decreasing proportion of 0.y.z packages, starting from April 2014. At that time, npm changed the initial version of packages created through `npm init` to 1.0.0 instead of 0.1.0.[17]

Based on this observation, we conjecture that policies and tools impact the proportions of 0.y.z packages in package distributions. For instance, the `cargo init` command to create new packages for Cargo sets the initial version of a new package to 0.1.0. Similarly, while gem, the official package manager for RubyGems, does not allow to create new packages, bundler, its recommended alternative,[18] sets the initial version of a newly created package to 0.1.0. On the other hand, composer, the official package manager for Packagist, does not specify a default initial version for newly created packages for Packagist since version numbers in Packagist are deduced from git tags.[19]

To understand to which extent the default initial version affects the considered package distributions, we looked at the version numbers adopted by the first public release of new packages. Table 5 reports about these version numbers and the proportion of packages created in 2019 that adopted them for their first public release.

We observe that Cargo and RubyGems, the two distributions that set the default initial version of a package to 0.1.0, have a much higher proportion

[17] See `https://github.com/npm/init-package-json/commit/363a17bc3`
[18] See `https://guides.rubygems.org/make-your-own-gem/`
[19] See `https://getcomposer.org/doc/articles/versions.md`

Table 5: Proportion of packages created in 2019 in function of the version number adopted for their first public release.

distribution	0.0.0	0.0.1	0.1.0	1.0.0	other 0.y.z	other ≥1.0.0
Cargo	13.5%	7.5%	55.3%	2.9%	19.6%	1.3%
RubyGems	4.7%	13.6%	47.1%	11.6%	14.3%	8.8%
npm	2.9%	18.0%	11.5%	40.8%	10.7%	16.1%
Packagist	0.6%	12.2%	13.3%	48.5%	7.5%	18.0%

of packages being released with a 0.1.0 initial version number. On the other hand, npm and Packagist have a much higher proportion of packages being released with a 1.0.0 initial version number. Specifically for npm, we computed the proportions for packages having been first distributed in 2013, the year preceding the adoption of 1.0.0 as the default initial version. We found that only 6.9% of packages were first released with 1.0.0, while 24% were released with 0.1.0, and 37.2% with 0.0.1. This seems to confirm that the adoption by npm of a new default value for the initial version of a package impacted the version number used by newly created packages.

However, while the adoption of this policy seems to have pushed maintainers to select a ≥1.0.0 version number for the initial version of their packages, nothing indicates that these packages are more mature or more stable than 0.y.z packages. Even if nothing in this policy prevents a package maintainer to choose a 0.y.z version number for the initial version of a package, a maintainer may also opt for a ≥1.0.0 version regardless of the maturity and stability of the package. Pushing developers to adopt a ≥1.0.0 version without clarifying the expectations and responsibilities associated with such versions is not an acceptable solution. It may prevent developers from communicating to dependent packages that their packages are, in fact, immature and unstable.

Lessons learned. Different package distributions adopt different versioning policies, explaining the observed differences for the proportion of 0.y.z packages.

Recommendations. Cargo and RubyGems should adapt their versioning policies to incite package maintainers to move out of the zero version space, the same way as npm has successfully done in 2014. However, package distributions should first clarify the expectations and responsabilities associated with a 1.0.0 version.

5.5. Semantic Versioning

Our findings revealed that the semver policy does not correspond to how Cargo, npm and Packagist deal with 0.y.z package releases in practice. This difference can be quite confusing for practitioners. Firstly, while semver considers

that "*major version zero is all about rapid development*", we found no conclusive evidence of this. Indeed, only a small proportion of packages crossed the 1.0.0 barrier, even after years of development. Moreover, we observed that 0.y.z releases are not updated considerably more frequently than ≥1.0.0 releases.

Secondly, semver has no specific rule dictating how to increment the version number of a 0.y.z release to indicate a compatible update. The policy is overly restrictive by assuming that "*anything may change at any time*" and that "*the public API should not be considered stable"*. Because of this, developers of packages adhering to semver cannot convey the backwards compatibility of 0.y.z releases through their version numbers. As a consequence, there is no way for a developer of a dependent package to declare a dependency constraint such that backwards compatible releases are allowed while at the same time preventing incompatible ones to be adopted. This means they have to decide either to face the risk of breaking changes, or to stay on the safe side by preventing the automatic installation of new 0.y.z releases and not being able to benefit from the bug and security fixes of these releases. Package distributions have therefore introduced notations and guidelines to circumvent this restriction. For example, Cargo defines caret requirements (i.e., ^x.y.z) as a way to "*allow semver compatible updates to a specified version*" but its implementation accepts patches for 0.y.z releases.[20] The same notation also exists in npm and Packagist and, although their semantics complies with semver for ≥1.0.0 releases, it does not for 0.y.z releases as it allows patches. Another example stems from the documentation of npm that recommends "*starting your package version at 1.0.0 to help developers who rely on your code*"[21] and even explicitly mentions that "*many authors treat a 0.x version as if the x were the "breaking-change" indicator*".[22] RubyGems is even more permissive since, while its official documentation "*urges gem developers to follow the Semantic Versioning standard*"[23], it makes no difference between 0.y.z and ≥1.0.0 when detailing how to increment version numbers with respect to backwards compatibility. This is a likely explanation for the findings in Section 4.5 that most dependencies towards 0.y.z releases accept new patches.

[20] `https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html`
[21] `https://docs.npmjs.com/about-semantic-versioning`
[22] `https://docs.npmjs.com/misc/semver`
[23] `https://guides.rubygems.org/patterns/#semantic-versioning`

Lessons learned. Package maintainers in the considered package distributions do not strictly follow semver for 0.y.z releases, and adopt a more permissive policy.

Recommendations. To fully benefit from semver, maintainers of mature packages should use ≥1.0.0 version numbers. Package distributions should explicitly document and communicate any deviation from the semver policy, or the semver policy should be adapted to allow maintainers to specify backwards compatible updates for 0.y.z releases.

5.6. Wisdom of the crowds

The semver policy also considers that "*if you have a stable API on which users have come to depend, you should be 1.0.0*" and that "*if your software is being used in production, it should probably already be 1.0.0"*. This is especially relevant in package distributions such as Cargo in which it is usual to have public dependencies as part of the API, as witnessed by a Rust developer: *"You may be ready to release 1.0 but if your crate reexports items from another and that crate hasn't released its 1.0 yet, then you're talking about entering a new level of dependency hell where you can't even upgrade dependencies without publishing a new major release*" [2].

However, our quantitative evidence reveals this semver guideline is not strictly followed in practice. Many 0.y.z packages are heavily used by other packages, including by "production-ready" (i.e., ≥1.0.0) packages. For example, the axios package on npm has not yet reached a ≥1.0.0 release, even though it is directly required by 30K other npm packages, and it exceeds 5M weekly downloads. A similar example for Cargo is the rand package. It has more than 25M downloads and more than 3K direct dependent packages, despite still being in 0.y.z since 2015 and having released more than 60 versions.

The fact that so many packages depend on 0.y.z packages suggests that these 0.y.z packages are more stable or more mature than what their version number suggests, hence reducing the risk of depending on them. The wisdom of the crowds principle has already been proposed to assess the backwards compatibility of packages based on the permissiveness of their (reverse) dependency constraints [6]. In a similar vein, developers desiring to depend on a 0.y.z package could rely on the number of existing dependents to assess the perceived maturity of that package. Package distribution managers could even go one step further, and provide automated support to suggest maintainers of such packages to upgrade their package version number to ≥1.0.0 based on this perceived maturity.

Lessons learned. Many 0.y.z packages are heavily used by other packages, even if semver signals it is risky to do so. This suggests that those packages are more mature than what their version number reflects.

Recommendations. Developers desiring to depend on 0.y.z packages could rely on wisdom of the crowds to assess the risk of doing so. Package distributions could use the same principle to provide automated support for recommending package maintainers to cross the 1.0.0 barrier.

6. Threats to Validity

We discuss the main threats that may affect the validity of our findings, following the structure recommended by Wohlin et al. [26].

Threats to *construct validity* concern the relation between the theory behind the experiment and the observed findings. The accuracy of our findings assumes that the package dependency metadata extracted from libraries.io is correct. We checked this assumption in previous work that relied on the same dataset [27, 6], by manually looking at hundreds of examples, as well as by comparing a subset of this dataset with the data available from the official package distributions (e.g., the npm package registry) or from GitHub.
Our findings may also be influenced by the "noise" present in the original data provided by the package distributions. As explained in Section 3, we removed such noise by excluding package releases from npm and Packagist that did not correspond to real development.
Another source of imprecision is caused by packages that changed their name over time. We believe this is an infrequent phenomenon, since the studied package distributions prohibit package renaming. Nevertheless, nothing prevents a maintainer from creating a package with a new name and abandoning the "old" package. In that case, the release history will be spread over two packages.

Threats to *internal validity* concern choices and factors internal to the study that could influence the observations we made. For RQ_6 we studied the influence of a range of characteristics for 0.y.z and $\geq$1.0.0 package repositories on GitHub. To provide additional insights, other relevant characteristics could have been included, such as the number of commits or number of pull requests.

Threats to *conclusion validity* concern the degree to which the conclusions we derived from our data analysis are reasonable. Given that our findings are based on empirical observations and on statistical tests with a high confidence level ($\alpha = 0.01$ adjusted after Bonferroni-Holm method to control family-wise error rate [22]), they are not affected by such threats.

The threats to *external validity* concern whether the results can be generalized outside the scope of this study. The proposed approach is certainly generalizable to other package distributions since it is mainly observational, as witnessed by the addition of RubyGems compared to our previous work [7]. The

observed findings themselves, however, are specific to the considered package distributions, since they are highly dependent on their policies, practices and tools. We already found important differences among the four package distributions we analysed, and we expect to see more such differences in other distributions, especially the ones relying on other versioning schemes (e.g., Hackage for Haskell or PyPI for Python).

7. Conclusion

In order for a mature software project to be considered healthy, it should avoid depending on unstable and immature reusable packages that are still in their initial development phase. A popular convention is to assume that 0.y.z package releases are more likely to be less complete, mature and stable than $\geq$1.0.0 package releases. This common belief is supported by anecdotal qualitative evidence that 0.y.z releases are perceived more risky and should be used with care. This belief is reflected in the versioning policies of package distributions that define different rules for 0.y.z and $\geq$1.0.0 package releases.

This paper aimed at finding quantitative evidence of how 0.y.z and $\geq$1.0.0 package releases behave in the Cargo, npm, Packagist and RubyGems package distributions. We observed that 0.y.z packages are prevalent in all four distributions, even contributing to more than 90% of all packages in Cargo. 0.y.z packages are as active as $\geq$1.0.0 packages, and they represent a large proportion of all package updates. We found that only a small proportion of packages went from a 0.y.z to a $\geq$1.0.0 release. While the majority of them took a few months and a few updates to do so, more than one out of five packages needed more than a year to reach a $\geq$1.0.0 release. We observed that 0.y.z releases are published slightly more frequently than $\geq$1.0.0 packages, but the difference is small in Packagist and RubyGems, and even negligible in Cargo and npm. We also found that the release frequency of most packages is higher for their 0.y.z releases than for their $\geq$1.0.0 releases.

We found that many 0.y.z packages are already used by other packages, and that many $\geq$1.0.0 packages are relying on 0.y.z packages. We studied how often 0.y.z and $\geq$1.0.0 releases are required by other packages but found no practical difference between them. We assessed whether 0.y.z releases comply with the semver policy by analysing dependency constraints towards 0.y.z releases. We found that the considered package distributions adopt a policy that is more permissive than semver, since most of these dependencies accept new patches. Finally, we found that a large majority of 0.y.z and $\geq$1.0.0 packages have an associated repository on GitHub. Repositories related to $\geq$1.0.0 packages tend to have slightly more stars, more forks, more contributors, more open issues, more dependent projects and are slightly larger than the ones related to 0.y.z packages, especially for RubyGems.

These quantitative findings go against the common wisdom that 0.y.z package releases should be used differently from $\geq$1.0.0 releases, as they are more likely to correspond to packages with a lower degree of maturity and stability usually associated to the initial development phase. This observed discrepancy

seems to come from the fact that many mature and production-ready packages remain stuck in the zero space. To fully benefit from semver they should be incited to upgrade to a $\geq$1.0.0 release. This can be done by package distribution managers, by adopting their versioning policies and by relying on wisdom of the crowds to detect which 0.y.z releases are likely to be stable and ready to move out of the zero space. Of course, the release number by itself is not sufficient to assess package maturity: there is a need for package distributions to clarify the expectations and responsibilities associated with distributing a package through the package manager.

The presented research can be extended in many ways. For example, one could rely on the development history of a package to assess at a fine level of granularity whether 0.y.z releases actually correspond to rapid development (e.g., based on the number and size of commits and code changes), contain less or less stable features (e.g., based on the number of feature and pull requests), or are more prone to bugs and security vulnerabilities (e.g., based on the number of reported issues). The presented quantitative analysis could be complemented by a full-fledged qualitative analysis based on in-depth interviews with package maintainers and users of these packages. Such interviews can help to understand why package maintainers are reluctant to cross the 1.0.0 barrier, how they perceive 0.y.z releases, and if they consider them different from $\geq$1.0.0 releases.

Acknowledgments

This work was supported by the Fonds de la Recherche Scientifique – FNRS under Grants number T.0017.18, O.0157.18F-RG43 and J.0151.20.

References

[1] J. Kahn, The Fear of 1.0.0 (2013).
URL http://jeremyckahn.github.io/blog/2013/12/29/the-fear-of-1-dot-0-0/

[2] Reddit, Don't fear 1.0.0! (2018).
URL https://www.reddit.com/r/rust/comments/9j6x9c/dont_fear_100/

[3] Reddit, The meaning of version 1.0.0 (2017).
URL https://www.reddit.com/r/rust/comments/7hr6ib/the_meaning_of_version_100/

[4] T. Preston-Werner, Semantic versioning 2.0.0 (June 2013).
URL https://semver.org

[5] C. Bogart, A. Filippova, C. Kästner, J. Herbsleb, F. Thung, Values and practices in 18 software ecosystems (2017).
URL http://breakingapis.org/survey/

[6] A. Decan, T. Mens, What do package dependencies tell us about semantic versioning?, IEEE Transactions on Software Engineering (2019) 1–1doi:10.1109/TSE.2019.2918315.

[7] A. Decan, T. Mens, How magic is zero? an empirical analysis of initial development releases in three software package distributions, in: ICSE Workshop on Software Health (SoHeal), 2020. doi:10.1145/3387940.3392205.

[8] V. T. Rajlich, K. H. Bennett, A staged model for the software lifecycle, IEEE Computer 33 (7) (2000) 66–71. doi:https://doi.org/10.1109/2.869374.

[9] A. Capiluppi, J. Gonzales-Barahona, I. Herraiz, G. Robles, Adapting the "staged model for software evolution" to free/libre/open source software, in: Int'l Workshop on Principles of Software Evolution, ACM, 2007, pp. 79–82. doi:https://doi.org/10.1145/1294948.1294968.

[10] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, A. Capiluppi, Empirical studies of open source evolution, in: Software evolution, Springer, 2008, pp. 263–288. doi:10.1007/978-3-540-76440-3_11.

[11] J. Costa, C. Chavez, P. Meirelles, On the sustainability of academic software: The case of static analysis tools, in: Brazilian Symposium on Software Engineering, ACM, 2018, pp. 202–207. doi:https://doi.org/10.1145/3266237.3266243.

[12] A. Stuckenholz, Component evolution and versioning state of the art, SIGSOFT Softw. Eng. Notes 30 (1) (2005) 7. doi:10.1145/1039174.1039197. URL https://doi.org/10.1145/1039174.1039197

[13] E. Wittern, P. Suter, S. Rajagopalan, A look at the dynamics of the JavaScript package ecosystem, in: Int'l Conf. Mining Software Repositories, ACM, 2016, pp. 351–361. doi:10.1145/2901739.2901743.

[14] S. Raemaekers, A. van Deursen, J. Visser, Semantic versioning and impact of breaking changes in the Maven repository, Journal of Systems and Software 129 (2017) 140 – 158. doi:https://doi.org/10.1016/j.jss.2016.04.008.

[15] A. Decan, T. Mens, M. Claes, An empirical comparison of dependency issues in OSS packaging ecosystems, in: Int'l Conf. Software Analysis, Evolution, and Reengineering, 2017, pp. 2–12. doi:10.1109/SANER.2017.7884604.

[16] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, How to break an api: cost negotiation and community values in three software ecosystems, in: International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 109–120. doi:https://doi.org/10.1145/2950290.2950325.

[17] J. Katz, Libraries.io open source repository and dependency metadata (version 1.6.0) (2020). doi:10.5281/zenodo.2536573.
URL https://doi.org/10.5281/zenodo.2536573

[18] O. Aalen, O. Borgan, H. Gjessing, Survival and Event History Analysis: A Process Point of View, Springer, 2008. doi:10.1007/978-0-387-68560-1.

[19] E. L. Kaplan, P. Meier, Nonparametric estimation from incomplete observations, Journal of the American Statistical Association 53 (282) (1958) pp. 457–481.
URL http://www.jstor.org/stable/2281868

[20] H. Hofmann, H. Wickham, K. Kafadar, Letter-value plots: Boxplots for large data, Journal of Computational and Graphical Statistics 26 (3) (2017) 469–477. doi:10.1080/10618600.2017.1305277.

[21] H. B. Mann, D. R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Statist. 18 (1) (1947) 50–60. doi:10.1214/aoms/1177730491.
URL https://doi.org/10.1214/aoms/1177730491

[22] S. Holm, A simple sequentially rejective multiple test procedure, Scandinavian Journal of Statistics 6 (2) (1979) 65–70.
URL http://www.jstor.org/stable/4615733

[23] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychological bulletin 114 (3) (1993) 494. doi:10.1037/0033-2909.114.3.494.

[24] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, L. Devine, Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices?, in: Annual Meeting of the Southern Association for Institutional Research, 2006.

[25] A. Decan, portion – python data structure and operations for intervals (2018).
URL https://github.com/AlexandreDecan/portion

[26] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012. doi:10.1007/978-1-4615-4625-2.

[27] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering (2018). doi:https://doi.org/10.1007/s10664-017-9589-y.