How Magic Is Zero?

An Empirical Analysis of Initial Development Releases in Three Software Package Distributions

Alexandre Decan
Tom Mens
alexandre.decan@umons.ac.be
tom.mens@umons.ac.be
Software Engineering Lab, University of Mons
Mons, 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, implying that they are likely to be less stable, and depending on them may be less healthy.

In this paper, we empirically study, for three open source package distributions (Cargo, npm and Packagist) to which extent 0.y.z package releases and ≥1.0.0 package releases behave differently. More specifically, 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, and we assess whether semantic versioning is respected for dependencies towards them. 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 be regarded as mature packages that are no longer under initial development. As a consequence, the version number does not provide a good indication of the health of a package release.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Reusability**; *Software evolution*; *Software version control*; *Maintaining software*.

SoHeal'20, May, 2020, Seoul, South Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/xxxxxxxxxx

KEYWORDS

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

ACM Reference Format:
Alexandre Decan and Tom Mens. 2020. How Magic Is Zero?: An Empirical Analysis of Initial Development Releases in Three Software Package Distributions. In *Proceedings of SoHeal 2020: International Workshop on Software Health (SoHeal'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/xxxxxxxxxx

1 INTRODUCTION

Open source software development embraces the principles of software reuse, through the availability of software package distributions dedicated to specific programming languages (e.g., Cargo for Rust, npm for JavaScript, 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. Packages that are under initial development assign a version number 0.y.z 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 ≥1.0.0 versions. If we assume that this convention is followed, it would be advisable for ≥1.0.0 packages to not depend on such 0.y.z packages, and for 0.y.z packages to quickly reach a ≥1.0.0 release in order to allow other packages to depend on a stable and mature release.

Some versioning policies explicitly materialise these differences between 0.y.z and ≥1.0.0 versions. Consider for example semver (semantic versioning) [15], a common versioning policy in package distributions [1], dictating how version numbers should be incremented w.r.t. backward compatibility. This policy distinguishes 0.y.z from ≥1.0.0 versions in terms of maturity, release cycle and stability. However, the specific rules for 0.y.z versions are sometimes considered disruptive[1] and counter-intuitive.[2] The term **magic zero** reflects this

[1] https://github.com/semver/semver/issues/221
[2] https://github.com/npm/node-semver/issues/79

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]

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 and Packagist, three package distributions that are known to adhere to semver [7]:

RQ_1**:** *How prevalent are 0.y.z packages?* We observe in all distributions that many packages did not yet reach a ≥1.0.0 release.

RQ_2**:** *How long does it take for a package to reach a ≥1.0.0 release?* Only a small proportion of packages traversed the 1.0.0 barrier, and one out of five took more than one year to do so.

RQ_3**:** *Are 0.y.z packages updated more frequently than ≥1.0.0 packages?* A statistical difference could be observed, but this difference was small to negligible in each distribution.

RQ_4**:** *Are 0.y.z package releases required by other packages?* This was indeed observed as a frequent phenomenon for each 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.[4]

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 presents the threats to validity of the research. Section 6 discusses the results and Section 7 concludes.

2 RELATED WORK

It is commonly accepted that the *initial development* stage of software should be distinguished from its subsequent evolution. This distinction is prominent in the staged software life cycle model [17]: "*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. [3], 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. [5] 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 [19]: "*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 [15] 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 [1]. Wittern et al. [20] 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. [16] 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. [8] 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. [7] 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 ecosystem-specific notations, characteristics, maturity and policy changes play an important role in the degree of such compliance.

Bogart et al. [2] 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

[3] https://github.com/npm/init-package-json/commit/363a17bc31bf653bb9575105eea62fb4664ad04b

[4] According to https://semver.org, *"Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable."*

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 ecosystems, on the basis of a survey involving more than 2,000 developers [1]. Different ecosystems 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 [7] and observed that three of these distributions (Cargo, npm, Packagist) 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 three package distributions.

To analyze the considered package distributions, we rely on version 1.4.0 of libraries.io Open Source Repository and Dependency Metadata [13], released in December 2018. 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*" [15]. 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). Since our focus is on how packages are actually being used, we exclude the dependencies that are only needed to test or develop the package, i.e., we only consider those dependencies that are required to install and execute the package. In the package distributions we analyzed, they are either labeled "runtime" or "normal".

To reduce noise in the dataset, we removed packages with clearly deviating and undesirable behaviour. For Packagist we excluded 21 of the most active packages (and their associated 1.2K releases) that were created and published to promote illegal download services. For npm we excluded around 22K packages (and their associated 50K 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.

distrib.	created	language	#pkg	#rel	#dep
Cargo	2014	Rust	21K	113K	433K
npm	2010	JavaScript	880K	5.979K	27.852K
Packagist	2012	PHP	141K	1.089K	3.079K

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.3693633.

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.

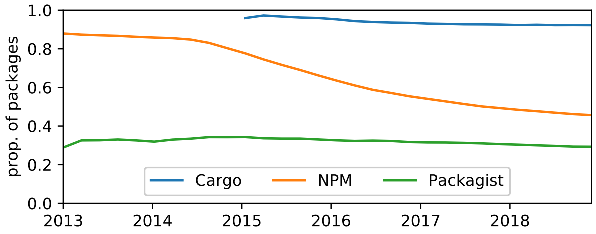

Figure 1: Evolution of the proportion of packages whose latest release is 0.y.z.

We observe a very high proportion of initial development packages for Cargo (94% on average), which is likely related to the young age of this package distribution. At the last observation time (December 2018), 92.2% of its packages were still 0.y.z packages. On the other side of the spectrum we find Packagist, a much older package distribution, with 32% of 0.y.z packages on average, and 29.3% at the last observation time. For npm, we observe from April 2014 onwards a decreasing proportion (from 85.5% to 45.7%) of 0.y.z packages. This is a consequence of npm policies aiming to reduce the use of 0.y.z releases.[5]

A possible explanation for these high observed proportions could be that many 0.y.z packages are no longer being maintained, preventing them from ever reaching a ≥1.0.0 release. To verify this, we repeated the analysis by removing all packages that were not active during the last 12 observed months (i.e., inactive in 2018). While this led to a decrease in the proportion of 0.y.z packages, the decrease remained limited: only 0.9% $(= 92.2 - 91.3)$ for Cargo, 5.7% $(= 29.3 - 23.6)$ for Packagist, and 8.3% $(= 45.7 - 37.4)$ for npm.

Summary. The considered package distributions contain many active 0.y.z packages: more than one out of five in Packagist, more than one out of three in npm, and more than nine out of ten in Cargo.

[5] See https://github.com/npm/node-semver/issues/79 and https://github.com/npm/init-package-json/commit/363a17bc3

4.2 How Long Does It Take to Reach ≥1.0.0?

If one assumes that 0.y.z packages are still in initial development, then they are eventually expected to reach a ≥1.0.0 release reflecting their maturation. We study whether this is indeed the case, and how long it takes in each considered package distribution. To do so, we distinguish three categories: 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 2 shows the proportion of these categories for each package distribution.

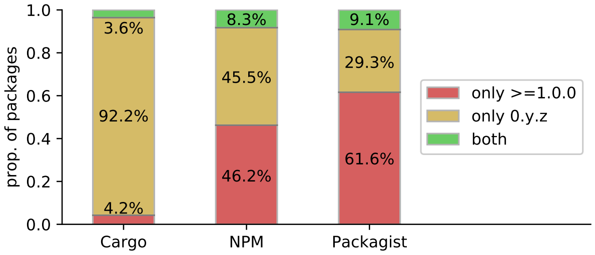

Figure 2: Proportion of packages having only ≥1.0.0 releases, only 0.y.z releases, or both.

We 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 738 packages in Cargo (3.6%), around 73K in npm (8.3%) and 13K packages in Packagist (9.1%). While most Cargo packages (92.2%) only have 0.y.z releases, the majority of Packagist packages (61.6%) 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.

Focusing 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 3 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).

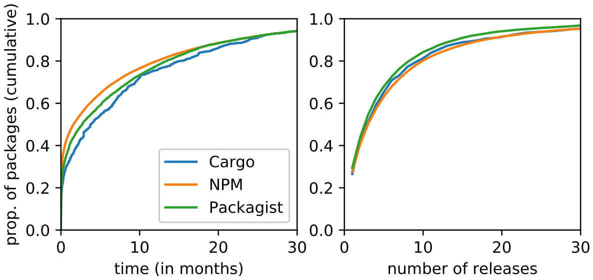

Figure 3: Duration (in months) and number of releases between first 0.y.z and first ≥1.0.0 release.

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 1.6 months (for npm) and 3.8 months (for Cargo) while the median number of release updates is 3 for each package distribution. Yet there are many packages that took much longer to reach ≥1.0.0. Over one out of five packages (24.7% in Cargo, 20.2% in npm and 22.6% in Packagist) needed more than a year to reach ≥1.0.0, and around 9% of packages in each package distribution needed more than 2 years.

Summary. Less than one out of ten 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 reach ≥1.0.0, and one out of ten took even more than two years.

4.3 Are 0.y.z Packages Updated More Frequently?

One would expect packages under initial development to release new updates 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, for 0.y.z and ≥1.0.0 releases respectively. Figure 4 shows the boxen plots [11] for these distributions.

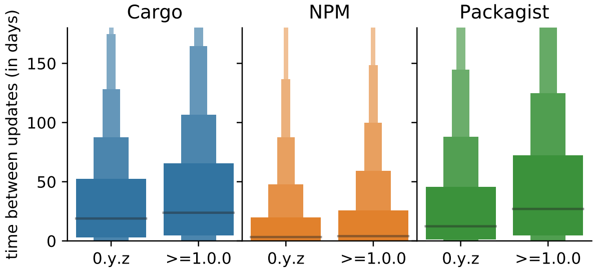

Figure 4: Distributions of the average time between consecutive releases, for 0.y.z and ≥1.0.0 releases.

We observe that for all package distributions, 0.y.z releases are more frequently updated 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, the median values are 18.9 and 23.8 days for Cargo (respectively for 0.y.z then ≥1.0.0 releases), 3.3 and 4 days for npm, and 12.6 and 27 days for Packagist. To confirm that these differences between 0.y.z and ≥1.0.0 release are statistically significant, we carried out Mann-Whitney-U tests [14]. The null hypothesis stating that there is no difference between 0.y.z and ≥1.0.0 releases was rejected for all three package distributions with $p < 0.01$ (adjusted after Bonferroni-Holm method to control family-wise error rate [12]). However, the effect size (measured using Cliff's delta $|d|$ [4]) revealed that the observed differences were *negligible* for Cargo ($|d| = 0.084$) and npm ($|d| = 0.027$), and *small* for Packagist ($|d| = 0.178$), following the interpretation of $|d|$ by Romanoet al. [18].

We also observe that the time to update a package release depends on the package distribution: package releases on npm are much more frequently updated than in the two other distributions. Mann-Whitney-U tests confirmed this observation with statistically significant differences ($p < 0.01$), implying that packages in npm are indeed updated more frequently than packages in Cargo and Packagist. In both cases the effect size was *small* ($|d| = 0.319$ for Cargo, $|d| = 0.324$ for Packagist).

Summary. 0.y.z package releases are updated more frequently than ≥1.0.0 releases, but the effect is small for Packagist, and negligible for Cargo and npm. Both 0.y.z and ≥1.0.0 package releases are updated more frequently in npm than in Cargo and Packagist.

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 [15]: "*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, considering the dependencies expressed in the latest release of each 0.y.z and ≥1.0.0 package. 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 (88.5% $= 81.8 + 6.7$) of the dependent packages in Cargo rely on 0.y.z releases, 81.8% being 0.y.z dependent packages as well and only 6.7% being ≥1.0.0 dependent packages. For Packagist, the inverse is true: most dependent packages (86.1% $= 21.3 + 64.8$) rely exclusively on ≥1.0.0 package releases. npm falls somewhere in the middle of both extremes, with 46.4% ($= 26.5 + 19.9$) dependent packages relying on 0.y.z package releases, and the remaining 53.6% ($= 21.1+32.5$) relying exclusively on ≥1.0.0 package releases. Nevertheless, in all three package distributions there is still a large number of dependent packages relying on at least one 0.y.z package.

When considering these numbers proportionally to the set of required packages (i.e., % targets), we observe that 87.6% ($= 75.9 + 11.7$) of the required packages in Cargo are 0.y.z packages. This proportion drops to 20% ($= 13.9 + 6.1$) for Packagist. Again, npm is in between, with 40.6% ($= 24.8 + 15.8$) of the required packages being 0.y.z packages. This indicates that in all three 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: package maintainers frequently depend on packages that are still under initial development, even though common wisdom says that such packages are more likely to be unstable.

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

		target			
ecosystem	**source**	**0.y.z**	**≥1.0.0**	**0.y.z**	**≥1.0.0**
Cargo	**0.y.z**	81.8	10.4	75.9	6.7
	≥1.0.0	6.7	1.2	11.7	5.7
npm	**0.y.z**	26.5	21.1	24.8	11.4
	≥1.0.0	19.9	32.5	15.8	48.0
Packagist	**0.y.z**	8.4	21.3	13.9	11.3
	≥1.0.0	5.5	64.8	6.1	68.7
proportionally to		**% sources**		**% targets**	

For each package distribution we computed the number of dependent packages (i.e., reverse dependencies) for 0.y.z and ≥1.0.0 packages, respectively. We compared both distributions using a Mann-Whitney-U test to find evidence of a statistical difference. The null hypothesis was rejected for all three package distributions ($p < 0.01$ after Bonferroni-Holm correction), suggesting that ≥1.0.0 packages are reused more. However, the effect size was *negligible* for all three package distributions ($0.062 \leq |d| \leq 0.097$).

Summary. Many packages are depending on 0.y.z packages, ranging from 13.9% of all dependent packages in Packagist to 88.5% in Cargo. Many 0.y.z packages are required by other packages, ranging from 20% of all required packages in Packagist to 87.6% in Cargo. We could not observe any practical difference between the number of dependent packages for 0.y.z and ≥1.0.0 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.*" [15]

When specifying package dependencies, package maintainers make use of *dependency constraints* to specify if new

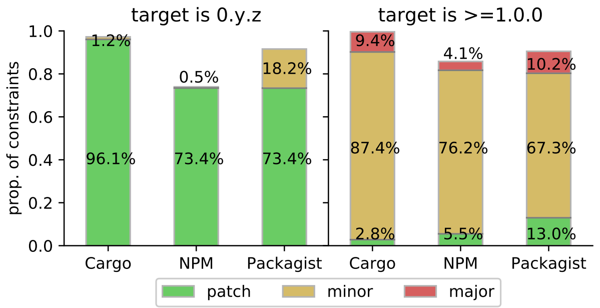

Figure 5: Proportion of dependency constraints accepting at most patches, minor or major releases, grouped by target releases.

patches, new minor and/or new major releases of the required package are allowed to be automatically installed and, by extension, are assumed to be backwards compatible. Since the way such constraints can be specified and interpreted depends on the package distribution [7], we wrote a dependency constraint parser for each package distribution to convert the constraints to a more generic and uniform version range notation [6].

Figure 5 reports the proportion of dependency constraints in the latest snapshot of each package distribution accepting at most patches, minor or major releases, separating between dependencies targeting 0.y.z releases and ≥1.0.0 releases, respectively.

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% (=73.4 + 0.5) in npm to 97.3% (=96.1 + 1.2) in Cargo. Packagist is even more permissive, since a non-negligible proportion of dependency constraints to 0.y.z package releases accept minor release updates as well (18.3% of all constraints targeting 0.y.z releases).

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 87.4% of such dependencies in Cargo, 76.2% in npm, and 67.3% in Packagist.

Summary. 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.

5 THREATS TO VALIDITY

We discuss the main threats that may affect the validity of our findings, following the structure recommended by Wohlin et al. [21].

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 manually checked this assumption in previous work that relied on the same dataset [7, 9]. Our findings also depend on the "noise" that may be 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 relates to the preparatory parsing step to convert dependency constraints to a more generic version range notations, as explained in Section 4.5. Since the large majority of constraints could be parsed (98.3%), this is unlikely to affect our results.

Threats to *internal validity* concern choices and factors internal to the study that could influence the observations we made. We did not find any such threats in our work.

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 [12]), 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. The observed findings themselves, however, are specific to the considered package distributions, since they are highly dependent on their policies and practices. We already found important differences among the three package distributions we analyzed, 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).

6 DISCUSSION

Our empirical analysis aimed to verify the convention that 0.y.z packages are considered to be under initial development and therefore potentially less stable than ≥1.0.0 releases. The results we obtained seem to suggest the opposite: most 0.y.z packages are assumed to be production ready and safe to use, implying there is probably little difference between how 0.y.z and ≥1.0.0 releases are perceived in practice.

The psychological 1.0.0 version barrier might explain why so few packages reach a ≥1.0.0 release. A 1.0.0 version is usually associated with the promise of a stable API and a mature library. We believe that most 0.y.z package developers 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.

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 pandas, one of the most famous Python libraries. Despite its widespread use in industry and academics, it will only reach its first ≥1.0.0 release during 2020, after more than 8 years of development and nearly one hundred releases.

Therefore, it is not surprising that we did not observe any fundamental difference between 0.y.z packages and ≥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 ≥1.0.0 releases, based on the presumed degree of maturity.

An alternative approach consists of not distinguishing between 0.y.z and ≥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*".[6] This is not a perfect solution either, since in practice it seems to encourage maintainers not to cross the 1.0.0 version barrier: "*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*".[7]

Summary. By defining different rules and conventions for 0.y.z and ≥1.0.0 releases, package distributions and versioning policies reinforce the artificial psychological barrier related to a 1.0.0 version number. There is no fundamental reason to consider that 0.y.z releases do not fulfil the same contracts or promises as ≥1.0.0 releases, especially as soon as a package is ready to be distributed and used by others.

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 went from 0.y.z to ≥1.0.0 releases, 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"*. Package distributions have therefore introduced notations and guidelines to circumvent this restriction: 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.[8] The documentation of npm recommends "*starting your package version at 1.0.0 to help developers who rely on your code*"[9] and even explicitly mentions that "*many authors treat a 0.x version as if the x were the "breaking-change" indicator*".[10] This is a likely explanation for the findings in Section 4.5 that most dependencies towards 0.y.z releases accept new patches.

Thirdly, semver 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"*. Our findings contradict these guidelines. We observed that 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.

Summary. Package maintainers in the considered package distributions do not strictly follow semver for 0.y.z releases, and adopt a more permissive policy. This deviation from the semver policy should be made explicit, or the semver policy should be adapted to allow maintainers to specify backwards compatible updates for 0.y.z releases.

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 consider that a 0.y.z version number corresponds to such initial development phase, conveying that the package is probably less complete, mature and stable than a ≥1.0.0 package release. This convention is reflected in package distributions and versioning policies that define different rules for 0.y.z and ≥1.0.0 package releases. In this paper, we verified if this convention is actually followed in practice, by empirically studying how 0.y.z and ≥1.0.0 package releases behave in the Cargo, npm and Packagist package distributions.

We observed that 0.y.z releases are prevalent in all three distributions, even contributing to 90% of all packages in Cargo. We found that only a small proportion of packages went from a 0.y.z to a ≥1.0.0 release. While the majority of them took a few months and a few updates to do so, one out of five packages needed more than a year to reach a ≥1.0.0 release.

[6] https://pvp.haskell.org/faq/

[7] https://www.reddit.com/r/haskell/comments/31e3jj/

[8] https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html

[9] https://docs.npmjs.com/about-semantic-versioning

[10] https://docs.npmjs.com/misc/semver

We observed that 0.y.z packages are updated slightly more frequently than $\geq$1.0.0 packages, but the difference is small in Packagist, and even negligible in Cargo and npm. 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. Finally, 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.

These results suggest there is probably little difference between how 0.y.z and $\geq$1.0.0 releases are perceived and behave in practice. They contradict the assumption that 0.y.z packages correspond to the lower degree of maturity and stability usually associated to the initial development phase, since in the packaging ecosystems we studied, many 0.y.z packages can already be considered as mature, stable and healthy packages.

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 qualitative one based on interviews of package developers. 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] Christopher Bogart, Anna Filippova, Christian Kästner, James Herbsleb, and Ferdian Thung. 2017. Values and practices in 18 software ecosystems. http://breakingapis.org/survey/

[2] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *International Symposium on Foundations of Software Engineering*. ACM, 109–120. https://doi.org/10.1145/2950290.2950325

[3] Andrea Capiluppi, Jesus Gonzales-Barahona, Israel Herraiz, and Gregorio Robles. 2007. Adapting the "staged model for software evolution" to Free/Libre/Open Source Software. In *Int'l Workshop on Principles of Software Evolution*. ACM, 79–82. https://doi.org/10.1145/1294948.1294968

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

[5] Joenio Costa, Christina Chavez, and Paulo Meirelles. 2018. On the sustainability of academic software: The case of static analysis tools. In *Brazilian Symposium on Software Engineering*. ACM, 202–207. https://doi.org/10.1145/3266237.3266243

[6] Alexandre Decan. 2018. *python-intervals 1.10.0 – Python data structure and operations for intervals.* https://github.com/AlexandreDecan/python-intervals

[7] Alexandre Decan and Tom Mens. 2019. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2918315

[8] Alexandre Decan, Tom Mens, and Maelick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *Int'l Conf. Software Analysis, Evolution, and Reengineering*. 2–12. https://doi.org/10.1109/SANER.2017.7884604

[9] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* (2018). https://doi.org/10.1007/s10664-017-9589-y

[10] Juan Fernandez-Ramil, Angela Lozano, Michel Wermelinger, and Andrea Capiluppi. 2008. Empirical studies of open source evolution. In *Software evolution*. Springer, 263–288. https://doi.org/10.1007/978-3-540-76440-3_11

[11] Heike Hofmann, Hadley Wickham, and Karen Kafadar. 2017. Letter-Value Plots: Boxplots for Large Data. *Journal of Computational and Graphical Statistics* 26, 3 (2017), 469–477. https://doi.org/10.1080/10618600.2017.1305277

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

[13] Jeremy Katz. 2018. Libraries.io Open Source Repository and Dependency Metadata (version 1.4.0). https://doi.org/10.5281/zenodo.2536573

[14] Henry B. Mann and Donald R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Ann. Math. Statist.* 18, 1 (03 1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[15] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. https://semver.org.

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

[17] Václav T. Rajlich and Keith H. Bennett. 2000. A Staged Model for the Software Lifecycle. *IEEE Computer* 33, 7 (July 2000), 66–71. https://doi.org/10.1109/2.869374

[18] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. 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*.

[19] Alexander Stuckenholz. 2005. Component Evolution and Versioning State of the Art. *SIGSOFT Softw. Eng. Notes* 30, 1 (Jan. 2005), 7. https://doi.org/10.1145/1039174.1039197

[20] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Int'l Conf. Mining Software Repositories*. ACM, 351–361. https://doi.org/10.1145/2901739.2901743

[21] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media. https://doi.org/10.1007/978-1-4615-4625-2