On the evolution of direct dependencies in npm packages

Shahin Ebrahimi-Kia[1], Jesus M. Gonzalez-Barahona[1], David Moreno-Lumbreras[1], Gregorio Robles[1] and Tom Mens[2]

[1] *Universidad Rey Juan Carlos, Madrid, Spain*
[2] *University of Mons, Belgium*

Abstract

Lehman's laws of software evolution postulate that software systems tend to grow in complexity and functional content. We aim to check if this growth can be observed when software applications are built as collections of reusable components, thus acknowledging that reused components contribute to growth and functional content. In particular, we study the evolution of the direct dependencies of JavaScript packages distributed through npm. We analyze this evolution with three different metrics capturing different growth patterns. Overall, we observe only small increases, suggesting that maintainers control the growth of the direct dependencies they rely on. We also find cases of increase/decrease patterns, which could signal specific efforts by package maintainers to reduce the number of dependencies.

Keywords

dependency analysis, software ecosystem, npm, Lehman's laws, software evolution, software complexity

1. Introduction

Package ecosystems of free, open source software (FOSS) components have revolutionized modern software development. Many software applications are no longer monolithic entities, but collections of software packages. Typically, the main application declares direct dependencies to packages to perform part of its functionality. In doing so, such applications have a new possible strategy for adapting to increasing complexity: instead of just incrementing their own code, they can rely on more packages [1]. In this work, we focus on the npm ecosystem of JavaScript/TypeScript packages because of its central role in web application development, and because it is the largest and fastest growing package ecosystem. For our analysis, we leverage on the fact that its public registry enables reproducible historical analysis.

Npm exhibits exponential growth [2], driven by the increasing reuse of packages to fulfill all kinds of functionality. This rapid growth resonates with Lehman's *laws of software evolution*, particularly those of *continuing growth*, *continuing change*, and *increasing complexity* [3]. These *laws* postulate that as software systems evolve, their complexity and interconnectedness naturally increases, demanding deliberate and continuous efforts to ensure stability and sustainability [4]. It remains an open question whether Lehman's insights also apply ecosystems like npm. Widespread availability of reusable packages makes the addition of direct dependencies a common maintenance action, which could make growth more likely, following *law of continuing growth*, which postulates that developers should constantly add new functionality to meet evolving requirements. But at the same time, the *law of increasing complexity* warns that uncontrolled dependency growth can lead to fragile applications, prone to cascading failures and difficult maintenance [3]. Ecosystems that fail to address these challenges risk accumulating technical debt, creating barriers to efficient software evolution [2].

In this paper, we study to what extent npm packages cope with complexity by adding new dependencies. We focus on *direct dependencies* because developers can control them directly: if they want to "delegate" functionality to a third-party component, they just add it as a new direct

shahin.ebrahimi@urjc.es (S. Ebrahimi-Kia); jesus.gonzalez.barahona@urjc.es (J. M. Gonzalez-Barahona); david.morenolu@urjc.es (D. Moreno-Lumbreras); gregorio.robles@urjc.es (G. Robles); tom.mens@umons.ac.be (T. Mens)

dependency. While we acknowledge that *indirect dependencies* drive much of the ecosystem risk (such as security vulnerabilities and breaking changes), they are not under direct control of developers. We therefore state as our main research question RQ_1: **Do npm packages increase their number of direct dependencies over time?**

To answer RQ_1 we analyze 1,998 npm packages drawn from four public importance-oriented lists, for reasons explained in Section 3. For the selected packages, we study their number of direct dependencies over time. Since we are interested in characterizing evolution, we also need to define metrics to decide if packages are increasing or decreasing in direct dependencies, leading to the auxiliary research question RQ_0: **How can we characterise changes in the number of direct dependencies of packages as they evolve over time?**

2. Related Work

The complex dynamics of dependency networks, especially in the npm ecosystem, has been extensively studied. Decan et al. [2] highlighted that npm's granular packaging and reuse introduce fragility and increased vulnerability risks through deeper dependency chains. Zerouali et al. [5] identified technical lag, where many packages fall behind due to delayed updates. Moreno-Lumbreras et al. [6] employed visualization, including VR, to explore npm's networks and management shortcomings. Prana et al. [7] revealed that tools like Dependabot alert outdated dependencies but often miss systemic risks like cascading failures. Zahan et al. [8] and Cogo et al. [9] studied supply chain weaknesses and dependency downgrades, respectively. Decan et al. [2] also discussed semantic versioning usage, noting inconsistencies leading to conflicts, while flexible dependency declarations aid update adoption but affect stability.

Research on npm package selection for reuse includes Mujahid et al. [10], who identified documentation quality, GitHub stars, and security as key factors; Abdalkareem et al. [11] and Qiu et al. [12] examined trivial package use and popularity metrics. Chatzidimitriou et al. [13] used network analysis to identify clusters, while Haefliger et al. [14] studied reuse patterns.

Security concerns are paramount, with Vu et al. [15] studying dependency management risks. Wheeler [16] proposed diverse double compiling to detect compiler injections. Kabir et al. [17] found widespread neglect of good practices, with vulnerable dependencies being common. Scalco et al. [18] proposed LastJSMile to detect source and artifact discrepancies. Zerouali et al. [19] analyzed outdated packages and risks, underscoring the need for software integrity in evolving dependency networks.

Dependency evolution impacts software integrity and verifiability. Harrand et al. [20] emphasized monoculture dangers, showing widely used libraries create security bottlenecks and amplify vulnerabilities, advocating diversity for resilience. Goswami et al. [21] found 38% of npm package versions unreproducible due to flexible versioning and tool variation.

Insights from other ecosystems provide context. Raemaekers et al. [22] showed Maven updates often break compatibility; Bavota et al. [23] mapped interdependencies affecting Apache projects. Germán et al. [24] revealed distinct evolution patterns in the R ecosystem, with core packages stable and peripherals more churned.

To date, systematic validation of Lehman's laws [25] in npm is lacking. Our work empirically examines if npm's dependency growth fits Lehman's laws, especially *continuing growth* and *increasing complexity*. Wittern et al. [26] documented npm's growth and intensifying dependency interconnectivity from 2011 to 2015, aligning with Lehman's *increasing complexity*. Lehman's early work used releases and modules as proxies for time and size. Later studies [3, 27, 28] continued this, with some simulating calendar time [29, 30]. Studies by Godfrey and Tu [31, 32] and Robles et al. [33] focused on size and calendar time. Israeli and Feitelson [34] expanded metrics for Linux evolution, adding complexity and maintainability indices [35]. We extend software growth metrics by focusing on direct dependencies.

3. Data Preparation

To perform our analysis, we create a historical dataset of package releases with their relevant dependency growth metrics, following the steps presented in the following subsections. The data used in this paper, along with the final results, and the software written to produce these results, are available in a reproduction package on Zenodo.[1]

3.1. List of considered packages

Instead of considering any random package in the npm registry, we create a *purposive sample* by focusing on packages which are "relevant to production deployments". To this end, we use the lists in npm Rank[2]. Since these lists date from 2019, they exclude newer packages, or packages that were not relevant at that time, which is discussed in Section 7). However, this cut-off time also ensures that most packages will have several years of history.

npm Rank provides four lists of 1,000 packages each:

1. **Top 1,000 most depended-upon packages:** Packages with the highest number of other packages directly depending on them.
2. **Top 1,000 packages with the largest number of dependencies:** Packages with extensive dependency trees, relying on numerous other packages for functionality.
3. **Top 1,000 packages with the highest PageRank score:** A ranking that considers both direct and indirect dependencies, based on a network analysis metric similar to Google's PageRank. Packages with high PageRank scores often play a central role in dependency networks.
4. **Top 1,000 packages with the highest authority:** Based on the Hyperlink-Induced Topic Search (HITS) algorithm, which distinguishes between "hubs" and "authorities" in a network. Studying these authoritative packages helps identify core libraries that are central to the stability and resilience of npm's ecosystem [36, 37].

We combined all lists in one, removing all duplicates, resulting in a set of 2,480 unique npm packages. To retrieve the relevant metadata for each of these packages, we used Open Source Insights API[3]. This metadata includes a list of the identifiers for all its releases in npm, and the number of direct dependencies for each of those releases. For 30 packages we were not able to obtain the metadata (e.g., because the package is no longer available), resulting in a final dataset of 2,450 packages.

3.2. Data filtering

Given this list of packages, we want to have into account releases that are "the most suitable for use in production at any given time". We decided to only consider releases with a three-component *semantic versioning* (SemVer) identifier[4], since those are usually recommended for production. We thus excluded non SemVer-compliant release identifiers with custom versioning schemas or non-numeric suffixes, such as pre-release and post-release identifiers suffixes (`alpha`, `rc`, `pre`, `post` and so on). At any point in time, we consider only the highest published release (following SemVer order), to consider only the "front-wave" versions, excluding backported releases.

Figure 1 illustrates this filtering of release identifiers for two packages, Express and Webpack. We show three types of releases: SemVer in blue, non-SemVer in orange, and backports to lower branches in purple. Our filtering retains only SemVer releases on the highest major branch.

[1] https://doi.org/10.5281/zenodo.15024304
[2] https://gist.github.com/anvaka/8e8fa57c7ee1350e3491 Details and code: https://github.com/anvaka/npmrank
[3] https://docs.deps.dev/api/v3/
[4] https://semver.org/

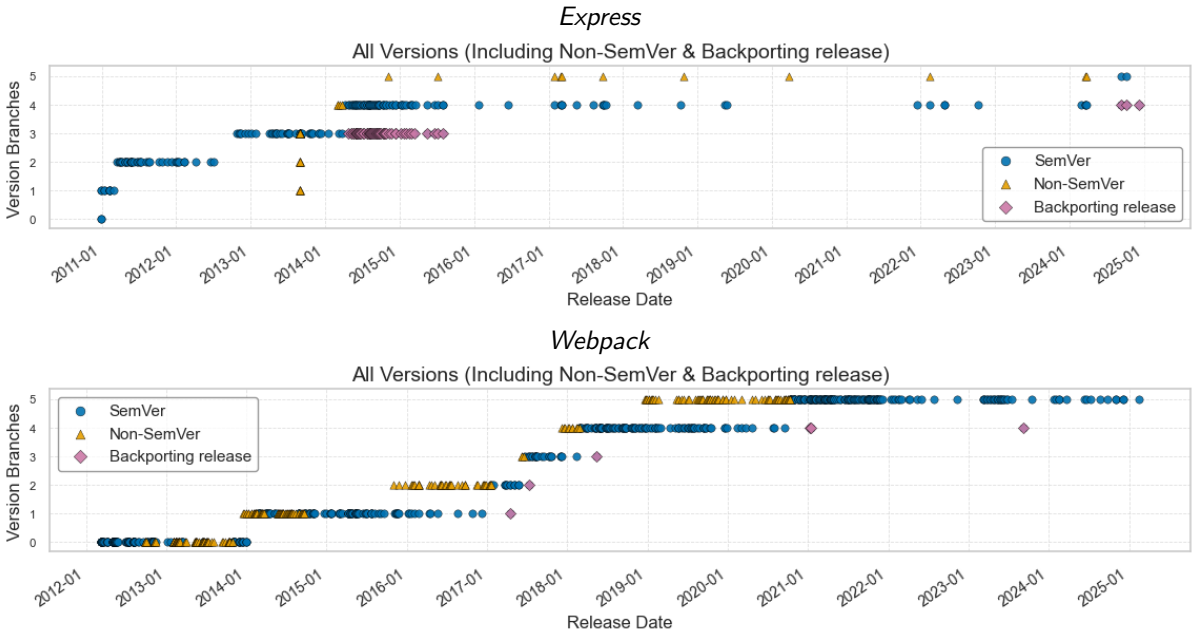
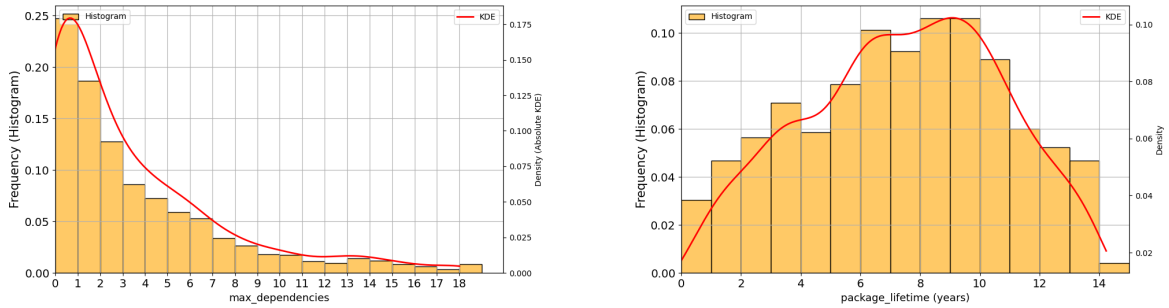


Figure 1: Identified release identifiers for `Express` (top) and `Webpack` (bottom). Each datapoint represents a release, with SemVer-compliant releases shown as blue circles, non-SemVer releases as orange triangles, and backporting releases as purple diamonds. This visualization clearly reveals the need to remove non-SemVer and backporting releases from the analysis.

After this filtering, we also excluded packages with less than six releases (452 packages), since those are too few to observe evolution. This resulted in a dataset of 1,998 packages and their SemVer-compliant release metadata and direct dependencies.



(a) Max direct dependencies.

(b) Package lifetime (years).

Figure 2: Distributions for the considered package dataset (outliers excluded).

Figure 2a and Figure 2b show two distributions of aimed at characterizing and better understanding our dataset. After removing outliers, to allow for a better visualization, Figure 2a shows the distribution of the maximum number of direct dependencies for each package. While one out of four packages has one direct dependency, and the majority of packages has less than four direct dependencies, there is a long tail of few packages having a larger number of dependencies. Figure 2b shows the distribution of the package lifetimes (time period from the first to the last release for each package). The peak is around 9 to 10 years, although most packages have shorter lifetimes.

3.3. Metrics definitions

Understanding how the number of direct dependencies evolves as new releases of a package are produced is not simple. There are many situations to consider: for some packages the number of dependencies increases, and then remains stable, or maybe decreases afterward, or decreases and then increases again. For some packages, dependencies grow or shrink quickly, for some others slowly. To take into account these nuances, we define multiple metrics, with the aim of characterizing the evolution of the growth of direct dependencies of a package release from different points of view.

To define these metrics, consider for each package release i a pair $(date_i, dep_i)$ in the Cartesian plane, where $date_i$ is the date of release i (represented in days since the date of the first release, i.e., $date_1 = 0$) and dep_i its number of direct dependencies. The pair corresponding to release 1 (the first one) is $(0, dep_1)$, and the pair for the last release n of the package is $(date_n, dep_n)$. Using this notation, we define the following metrics:

Absolute Change $= dep_n - dep_1$ Difference of direct dependencies between the last and the first release. This characterizes the net increase or decrease in number of dependencies.

First-Last Slope $= \frac{dep_n - dep_1}{date_n}$ Slope of the straight line from the pair corresponding to the first release, to the pair corresponding to the last release of the package. It measures the growth from the first release to the last one, taking into account the time span between them. Positive or negative values for this metric indicate an increase or decrease in dependencies. The larger numbers represent steeper increases or decreases.

Linear Slope $= \frac{\sum_{i=1}^{n}(date_i - \bar{date})(dep_i - \bar{dep})}{\sum_{i=1}^{n}(date_i - \bar{date})^2}$ ($\bar{x}$ stands for the mean of the distribution $x_1 \dots x_n$) Slope of the straight line resulting of performing a linear regression fit for the pairs corresponding to all releases of the package. This captures the overall trend, taking into account all releases.

Growth Fraction Fraction of the time during which a polynomial fit for all the pairs grows. We use a second-order polynomial regression, for characterizing the cases when there is a mix of increase / decrease periods with a single number which approximates the fraction of time dependencies are growing. The fitting is therefore a function of the form $y = ax^2 + bx + c$. Its derivative $y' = 2ax + b$ shows when the function increases or decreases, and when it changes from one to the other. This metric is computed as the period during which the derivative is positive, normalized between 0 and 1. 0 is for the case when there is no positive derivative, therefore the number of dependencies never grows, and 1 is for the case when the derivative is positive during the entire package lifetime, which means dependencies are always growing.

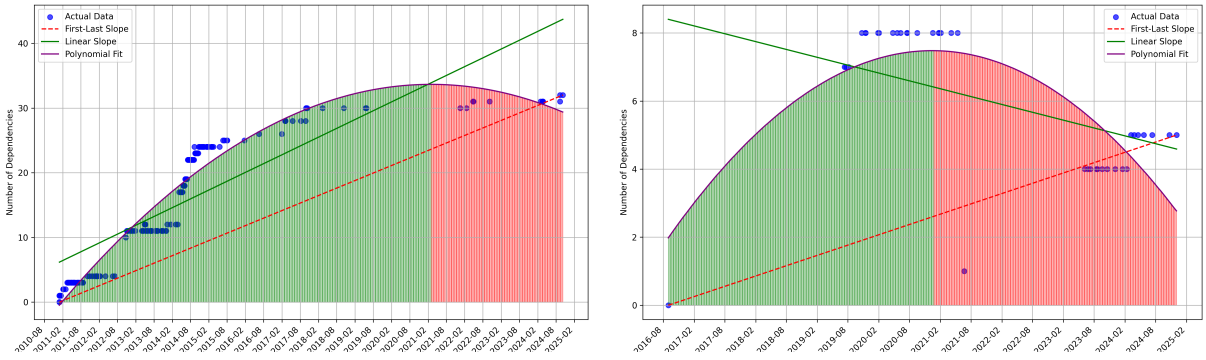
Package	Absolute Change	First-Last Slope	Linear Slope	Growth Fraction
`Express`	32	0.006	0.007	0.7350
`Web3-utils`	8	0.002	-0.001	0.5200

Table 1
Growth metrics for packages `Express` and `Web3-utils`.

Let us illustrate these metrics on our two example packages. All metrics values are reported in Table 1. Metrics values for other specific packages are available in the reproduction package.

Figure 3 shows that `Express` experiences a steady increase in direct dependencies, up to 2018, when it flattens. This is not captured by *Absolute Change*, *First-Last Slope*, or *Linear Slope*, which all just show increase. *Growth Fraction*, in this case, captures the two phases in the

growth, but not exactly right: it signals correctly an increase up to 2021, but then it shows a decrease which is not really happening. However, it clearly shows that the trend is not just steady growth, as the other metrics show. `Web3-utils` starts increasing its dependencies (from 0 to 8) until later 2019, then remains stable, steps down sharply in mid 2021, and finally grows again. *Absolute Change* and *First-Last Slope* only increase, while *Linear Slope* shows decrease in dependencies. *Growth Fraction* captures the situation better, showing two phases, with a decrease in the second which, even though not real, is still correct if we compare the plateau around 2020 to the final years.



(a) `Express` (b) `Web3-utils`

Figure 3: Visualisation of the elements used to calculate the four dependency growth metrics for two packages.

4. Analysis Results of the Dependency Growth Metrics

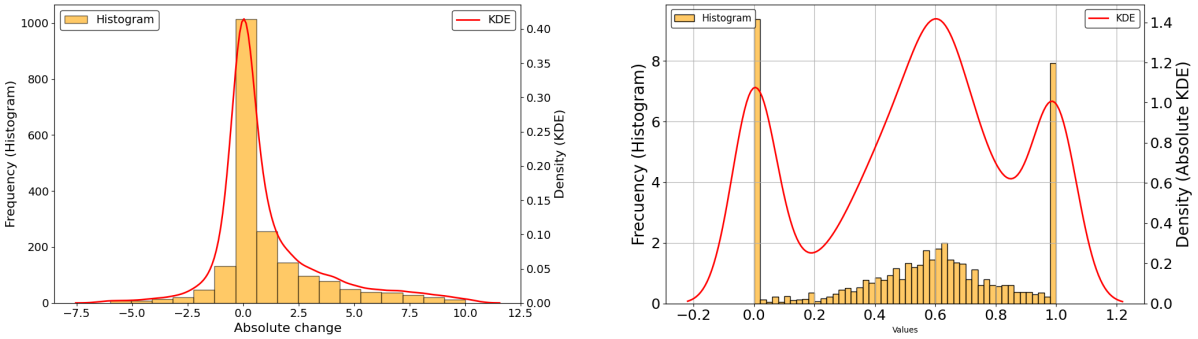
This section present the results of computing the growth metrics on the entire dataset. The descriptive statistics of all metrics are summarised in Table 2.

Table 2
Descriptive statistics for the four growth metrics for all considered packages

Metric	Mean	Median	Std. Dev.	Min	Max	Q1 (25%)	Q3 (75%)
Absolute Change	8.5143	0	27.9878	-310	462	0	4
First-Last Slope	0.15190	0	2.2360	-44.286	65.5	0	0.003
Linear Slope	0.16595	0	1.8994	-13.676	46.786	0	0.003
Growth Fraction	0.53288	0.5775	0.33468	0	1	0.315	0.775

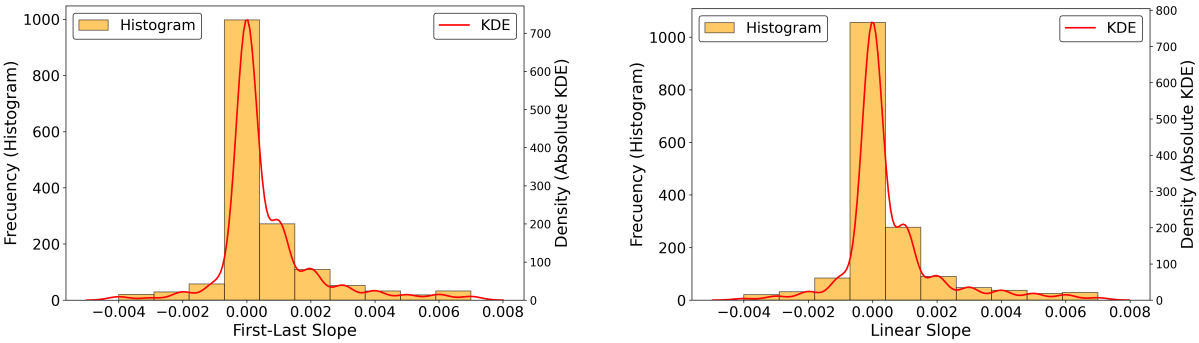
To avoid distortion from extreme values, for all the charts below we applied interquartile range (IQR) filtering to remove outliers. This method removes values outside 1.5 times the middle 50% range, excluding extreme data while preserving typical behavior for accurate trends.

Absolute Change Figure 4(a) shows a histogram and a KDE of the distribution of *Absolute Change* values for all packages, after removing some outliers to make the visualization more clear. The distribution is centered around zero, with some skewness toward positive numbers (median is 0, as well as Q1, but mean is close to 8.5, and Q3 is 4). The most populated bucket is the one with a metric of 0 (more than 1,000 packages, or about half of all packages). The minimum and maximum observed values are -310 and 462. According to this metric, packages tend not to grow or shrink in number of direct dependencies, or do it very slightly. Most of them are quite grouped around a variation of 0 (the same number of dependencies at the start and at the end of the project). However, packages that grow tend to grow more than those that shrink. Some packages have huge variations in the number of dependencies, in the order of hundreds of them.



(a) Absolute Change (b) Growth Fraction

Figure 4: Histograms and KDEs of Absolute Change and Growth Fraction for all packages (excluding outliers).



(a) First-Last Slope. (b) Linear Slope.

Figure 5: Histograms and KDEs of First-Last Slope and Linear Slope for all packages (excluding outliers).

First-Last Slope Figure 5(a) shows a histogram and a KDE of the distribution of *First-Last Slope* values for all packages, after removing some outliers. The distribution is very much concentrated around zero (Q1 and median are 0, Q3 is almost 0 as well), very slightly skewed towards positive numbers. The central bucket, with slopes close to 0, is by far the larger. There are some outliers with extreme slopes (less than -44 and more than 65 for packages with decreasing and increasing dependencies, respectively). This metric shows even more stability in the number of dependencies per package than the previous one. According to it, we could say that, by far, most packages remain stable over time, even when some of them grow or shrink, and only very few of them considerably. The growth from the first to the last release tends very strongly to zero.

Linear Slope Figure 5(b) shows a histogram and a KDE of the distribution of *Linear Slope* values for all packages, after outlier removal. The distribution is very much concentrated around zero, with Q1, Q3, and median 0 or almost 0, very slightly skewed towards positive numbers. The central bucket, with slopes close to 0, is by far the largest. There are some outliers with extreme slopes (less than -13 and more than 46 for packages with decreasing and increasing dependencies, respectively). There is a bit less dispersion than for *First-Last Slope* (standard deviation of 1.9 versus 2.2). Once again, this metric mainly shows that packages are stable in the number of their direct dependencies. There is even more concentration close to 0, with some extreme outliers. We can say that linear growth of direct dependencies tends very strongly to zero.

Growth Fraction Figure 4(b) shows a histogram and a KDE of the distribution of *Growth Fraction* values for all packages, after outlier removal. The distribution has two groups con-

centrated in the extremes (0 and 1), and then another one, much lower, point of concentration around 0.6. The distribution seems a bit skewed towards 1 (median is 0.58, mean is 0.53), and it is relatively symmetrical. According to this metric, packages seem to spend more time growing than shrinking in number of direct dependencies. However, there are more packages that reduce their dependencies than packages that increase them, if we consider only those who either reduce or increase dependencies during all their life, respectively (values of the metric of 0 or 1).

5. Other observations

Our results show that, in general, packages tend to keep their number of direct dependencies stable. For cases with clear growth or decline, we investigated whether certain parameters influence the likelihood of increasing or decreasing dependencies. We focused on the *Linear Slope* metric to capture trends in dependency changes. We explored its relationship with maximum dependencies, package lifetime, and number of releases per package. Table 3 presents descriptive statistics for subsamples based on these criteria. Classification is based on median values: below median are "low", "short" or "few"; at or above median are "high", "long" or "many".

Table 3
Descriptive statistics of *Linear Slope* for specific populations of packages

Metric	Mean Slope	Median Slope	Std. Dev.
few Max Dependencies	0.0002	0	0.0009
many Max Dependencies	0.3386	0.003	2.7037
short Lifetime	0.3678	0.001	2.8225
long Lifetime	0.0015	0	0.0071
low Number of Releases	0.3418	0	2.8393
high Number of Releases	0.0263	0.001	0.1744

Packages with many dependencies show a higher mean slope (0.3386) than those with fewer (0.0002), indicating that more complex packages undergo more dependency changes. Similarly, packages with a short lifetime have a higher mean slope (0.3678) than long-lived packages (0.0015), suggesting that shorter-lived packages experience more dynamic dependency shifts. Finally, packages with a low number of releases exhibit a higher mean slope (0.3418) than those with a high number (0.0263), meaning that infrequently updated packages have more drastic variations. The median slopes remain close to zero across categories, showing that most packages maintain stable dependencies. In summary, we can state that complex, short-lived, and infrequently updated packages more often experience notable dependency shifts.

We also examined how the number of direct dependencies per package evolves over time. For this, we computed the mean and median number of direct dependencies for all packages at the end of every year. We considered, for each package, the latest available release at the end of the year (in other words, the latest release for that package that could be deployed at the end of each year). If the package didn't produce any release by that date, it is not considered. The results of this analysis are presented in Figure 6.

The figure shows a rapid increase in mean number of dependencies between 2015 and 2019, followed by a plateau starting around 2019. This plateau is likely to be influenced by the composition and timing of our dataset, which was produced according to the situation in 2019. Thus, packages that are "relevant" after 2019 are mist likely underrepresented, which could explain why the increasing trend does not continue after 2019.

Several phases can be identified in the evolution of the mean number of direct dependencies per package: *slow growth* until 2014, when both mean and median remain close to zero; *expansion* until 2019, with a sharp increase in mean direct dependencies, while median growth happens much more slowly; and *plateau* until the end of the time period, with both metrics remaining

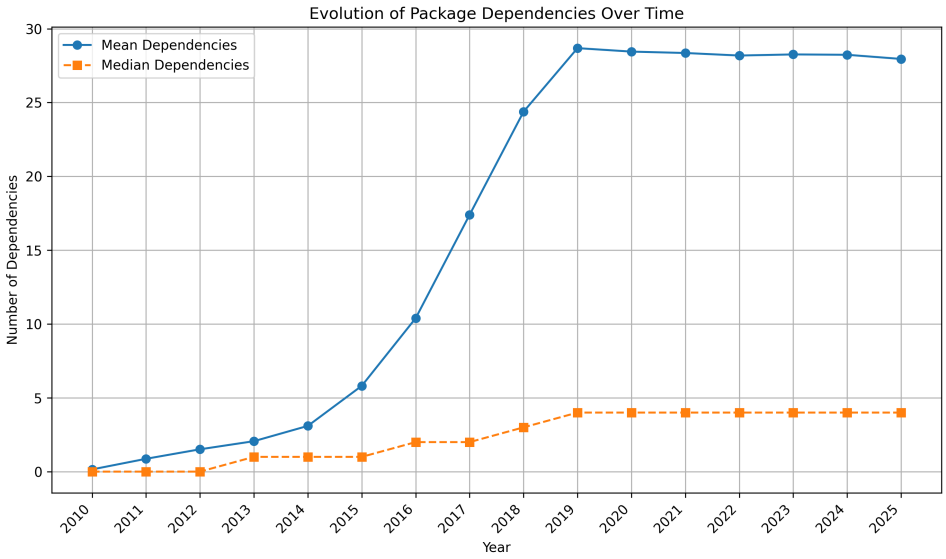


Figure 6: Mean and median number of direct dependencies over time.

stable. In addition to the growth patterns, it is worth noticing that the increase in the difference between mean and median signals an increase in the spread of the distribution, with some packages evolving towards a very high number of dependencies.

6. Discussion

We answered RQ_0 by defining four metrics, designed to capture different aspects of the growth of direct dependencies, acknowledging the fact that there are many different growth patterns, and when one metric may identify growth, another one may identify decline. However, when we aggregate all packages, the results of these metrics mostly align. The two slope-based metrics have very similar descriptive statistics, and the other two show consistent distributions.

This allows us to answer RQ_1 in a conclusive way: the growth in number of direct dependencies over time in the considered packages is positive, but very close to zero. Therefore, Lehman's laws do not seem to apply to this kind of growth for our sample of packages. If JavaScript applications are growing, they will be in different ways (e.g. in terms of code size, or in terms of transitive dependencies, which have not been considered in this paper).

Of course, this doesn't mean that there are packages with steep increases or declines in the number of direct dependencies. We found some related parameters which could be a predictor for those cases, the most important of them being the age of the project (in number of releases or in time): young packages tend to grow, and grow faster.

From the point of view of developers, direct dependencies can be controlled directly by package maintainers. Therefore, their growth or decline falls completely under their responsibility. The results in this paper may help them to have a benchmark for comparison, so that they can analyze the evolution of their direct dependencies in the context of what is happening in comparable packages. In fact, this was one of the reasons why we included packages "relevant to production" in the dataset, instead of taking a random sample of packages: we want our study to be useful for practitioners, and for that we need to let them compare their components with relevant packages.

7. Threats to validity

Construct validity. We defined four metrics to quantify the temporal evolution of direct dependencies. Those metrics may not capture growth adequately, or may mask other effects which would lead to incorrect conclusions. However, we defined these metrics to capture different aspects of dependency growth, considering that the number of direct dependencies tends to be relatively low.

For measuring package growth we considered only releases with the highest SemVer identifier when there was more than one development branch. This aims to capture the "front wave" of stable development, but may inaccurately reflect more complex development processes and usage patterns. It could happen, for example, that for long periods of time, new branches are ignored by users because they are still unstable, despite developers tagging them as stable releases. Future work could consider other ways of selecting the releases to measure growth.

We can also not discard errors in the data source we used for retrieving packages, and in the tools and scripts that we used for the analysis.

Internal validity. We selected the packages for building our base dataset based on several metrics of 'relevance'. But those metrics could not really represent relevance, or maybe relevance is not really relevant for this kind of study. Besides, the list is 5 years old, which means that the relevance of those packages may have shifted since then. Finally, the list is maybe too short fo be representative of 'relevant' packages. However, a manual inspection of it show many packages very relevant in the current software development, and the age of the list also allowed us to have packages with a certain lifetime, important to being able of measuring evolution in them.

We built our base dataset following a filtering process, trying to capture the real evolution of a package, removing all pre-release releases, and considering only versions "intended for deployment in production". We consider that our filtering strategy ensures that the dataset remains reliable and representative of real-world dependency trends, minimizing bias by eliminating unstable releases and outdated branches. But maybe this does not capture well the deployment practices of developers.

External validity. We cannot claim that all packages in any ecosystem behave the same way as the packages in our base dataset. In fact, we cannot even claim that in the case of npm, all packages behave like ours. We selected a very specific sample, trying to include in it packages relevant for production environments. From this point of view, even when we don't know if the patterns that we have found are common in other collection of packages, we think we can claim that we used a good sample of packages with an industrial interest, and therefore our results are likely useful for other similar packages, at least in the npm ecosystem.

Of course, our analysis is limited in time. Even when the data is very recent (from March 2025), the list we are using is from 2019. It could happen that the evolution of the npm ecosystem is such, that our results are no longer valid for current 'relevant' packages.

8. Conclusion

To analyze how direct dependencies of npm packges evolve over time, we defined several metrics to consider different aspects of their growth. We measured them on a curated collection of npm packages, created with the main aim of including those relevant for production.

Understanding this evolution helps to understand to which extent Lehman's laws apply to applications built by composing packages, in which developers balance growth by adding more code of their own with growth by adding dependencies on reusable components.

Our findings reveal that, while most packages exhibit stability in dependency counts, a small subset shows either rapid growth or decline, maybe indicating shifts in software development practices. These variations highlight the diversity in software evolution, where some packages experience dependency expansion, while others undergo simplification or deprecation.

References

[1] M. Vieira, D. Richardson, The role of dependencies in component-based systems evolution, in: International Workshop on Principles of Software Evolution, 2002, pp. 62–65.

[2] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering 24 (2018) 381–416. doi:10.1007/s10664-017-9589-y.

[3] M. M. Lehman, Laws of software evolution revisited, in: European Workshop on Software Process Technology, Springer, 1996, pp. 108–124.

[4] I. Herraiz, D. Rodriguez, G. Robles, J. M. Gonzalez-Barahona, The evolution of the laws of software evolution: A discussion based on a systematic literature review, ACM Computing Surveys (CSUR) 46 (2013) 1–28.

[5] A. Zerouali, T. Mens, J. M. Gonzalez-Barahona, G. Robles, A formal framework for measuring technical lag in component repositories—and its application to npm, Journal of Software: Evolution and Process 31 (2018) e2157. doi:10.1002/smr.2157.

[6] D. Moreno-Lumbreras, J. M. González-Barahona, M. Lanza, Understanding the NPM dependencies ecosystem of a project using virtual reality, in: Working Conference on Software Visualization, 2023, pp. 84–92. doi:10.1109/VISSOFT60811.2023.00019.

[7] G. A. A. Prana, C. Bird, E. T. Barr, P. T. Devanbu, A. Hindle, Using practitioners' insights to investigate reproducibility of software engineering studies, Empirical Software Engineering 26 (2021) 1–37.

[8] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, L. Williams, What are weak links in the npm supply chain?, in: International Conference on Software Engineering, 2022, pp. 331–340.

[9] F. R. Cogo, G. A. Oliva, A. E. Hassan, An empirical study of dependency downgrades in the npm ecosystem, IEEE Transactions on Software Engineering 47 (2019) 2457–2470.

[10] S. Mujahid, R. Abdalkareem, E. Shihab, What are the characteristics of highly-selected packages? a case study on the npm ecosystem, Journal of Systems and Software 198 (2023) 111588.

[11] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, E. Shihab, Why do developers use trivial packages? an empirical case study on npm, in: Joint Meeting on Foundations of Software Engineering, 2017, pp. 385–395.

[12] S. Qiu, R. G. Kula, K. Inoue, Understanding popularity growth of packages in JavaScript package ecosystem, in: International Conference on Big Data, Cloud Computing, Data Science and Engineering (BCD), 2018, pp. 55–60. doi:10.1109/BCD2018.2018.00017.

[13] K. C. Chatzidimitriou, M. D. Papamichail, T. Diamantopoulos, N.-C. I. Oikonomou, A. L. Symeonidis, npm packages as ingredients: A recipe-based approach., in: ICSOFT, 2019, pp. 544–551.

[14] S. Haefliger, G. Von Krogh, S. Spaeth, Code reuse in open source software, Management Science 54 (2008) 180–193.

[15] I. Pashchenko, D.-L. Vu, F. Massacci, A qualitative study of dependency management and its security implications, in: ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 1513–1531.

[16] D. A. Wheeler, Countering trusting trust through diverse double-compiling, in: Annual Computer Security Applications Conference (ACSAC), 2005, pp. 13–48. doi:10.1109/CSAC.2005.17.

[17] M. M. A. Kabir, Y. Wang, D. Yao, N. Meng, How do developers follow security-relevant best practices when using NPM packages?, in: Secure Development Conference, 2022, pp. 77–83. doi:10.1109/SecDev53368.2022.00027.

[18] S. Scalco, R. Paramitha, D.-L. Vu, F. Massacci, On the feasibility of detecting injections in malicious npm packages, in: International Conference on Availability, Reliability and Security, 2022, pp. 1–8.

[19] A. Zerouali, V. Cosentino, T. Mens, G. Robles, J. M. Gonzalez-Barahona, On the impact of outdated and vulnerable JavaScript packages in Docker images, in: International Conference on Software Analysis, Evolution and Reengineering, 2019, pp. 619–623. doi:10.1109/SANER.2019.8667984.
[20] N. Harrand, Software Diversity for Third-Party Dependencies, Ph.D. thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2022.
[21] P. Goswami, S. Gupta, Z. Li, N. Meng, D. Yao, Investigating the reproducibility of NPM packages, in: International Conference on Software Maintenance and Evolution, 2020, pp. 677–681. doi:10.1109/ICSME46990.2020.00071.
[22] S. Raemaekers, A. Van Deursen, J. Visser, The Maven repository dataset of metrics, changes, and dependencies, in: Working Conference on Mining Software Repositories, IEEE, 2013, pp. 221–224.
[23] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, S. Panichella, The evolution of project inter-dependencies in a software ecosystem: The case of Apache, in: 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 280–289.
[24] D. M. German, B. Adams, A. E. Hassan, The evolution of the R software ecosystem, in: European Conference on Software Maintenance and Reengineering, IEEE, 2013, pp. 243–252.
[25] M. M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68 (1980) 1060–1076. doi:10.1109/PROC.1980.11805.
[26] E. Wittern, P. Suter, S. Rajagopalan, A look at the dynamics of the JavaScript package ecosystem, in: International Conference on Mining Software Repositories, ACM, 2016, pp. 351–361. doi:10.1145/2901739.2901743.
[27] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and laws of software evolution—the nineties view, in: International Software Metrics Symposium, IEEE, 1997, pp. 20–32.
[28] M. M. Lehman, J. F. Ramil, An approach to a theory of software evolution, in: International Workshop on Principles of Software Evolution, 2001, pp. 70–74.
[29] B. W. Chatters, M. M. Lehman, J. F. Ramil, P. Wernick, Modelling a software evolution process: A long-term case study, Software Process: Improvement and Practice 5 (2000) 91–102.
[30] P. Wernick, M. M. Lehman, Software process white box modelling for FEAST/1, Journal of Systems and Software 46 (1999) 193–201.
[31] Q. Tu, M. Godfrey, Evolution in open source software: A case study, in: International Conference on Software Maintenance, IEEE, 2000, pp. 131–142.
[32] M. Godfrey, Q. Tu, Growth, evolution, and structural change in open source software, in: International Workshop on Principles of Software Evolution, 2001, pp. 103–106.
[33] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, I. Herraiz, Evolution and growth in large libre software projects, in: International Workshop on Principles of Software Evolution (IWPSE), IEEE, 2005, pp. 165–174.
[34] A. Israeli, D. G. Feitelson, The Linux kernel as a case study in software evolution, Journal of Systems and Software 83 (2010) 485–501.
[35] D. Coleman, D. Ash, B. Lowther, P. Oman, Using metrics to evaluate software system maintainability, Computer 27 (1994) 44–49.
[36] C. Ridings, M. Shishigin, PageRank Uncovered, Technical Report, Technical report, 2002.
[37] L. Page, S. Brin, R. Motwani, T. Winograd, The PageRank Citation Ranking: Bringing Order to the Web, Technical Report, Stanford Digital Libraries, SIDL-WP-1999-0120, 1999.