Does Software Still Grow? An Empirical Study of Lehman's Laws in npm Packages

Shahin Ebrahimi-Kia[1], Jesús M. Gonzalez-Barahona[1], David Moreno-Lumbreras[1] and Gregorio Robles[1]

[1]*Universidad Rey Juan Carlos, Madrid, Spain*

Abstract

We summarize a study submitted to the Journal of Systems and Software that examines Lehman's Law of Continuing Growth through 155,532 releases of 2,091 popular npm packages. Depending on the growth metric, 57.5%–64.2% of packages grow in total code size. Internal-code growth is more widespread than dependency-code growth, although the latter becomes important in dependency-dominated packages. These findings are consistent with continuing growth and highlight the need to distinguish internal code from external dependencies.

Keywords

software evolution, Lehman's laws, npm, dependencies, software growth

1. Motivation and research questions

Lehman's Law of Continuing Growth states that functional content must increase to maintain user satisfaction [1]. In component-based development, functionality can come from internal code or independently maintained dependencies. Ecosystem-level dependency growth [2] need not imply growth in individual packages.

We ask how popular production-ready npm packages evolve in size (RQ1), how internal and external code contribute (RQ2), and whether package subgroups differ (RQ3). We use lines of code (LOC) as a size proxy and dependency counts for context, not as a comprehensive complexity measure.

2. Dataset and method

Using npm metadata queried on March 17, 2026, we selected 2,466 packages with at least 500 dependents and one million weekly downloads. We retained stable SemVer releases from version 1.0.0 onward and excluded later releases of older major–minor branches once a newer branch was available. Requiring at least five retained releases spanning more than one year produced 2,091 packages. Unavailable package contents excluded 22 releases, leaving 155,532 measured releases across all 2,091 packages.

For each release, we recursively resolved production dependencies using the highest version satisfying each constraint and published no later than that release. Each resolved package–version pair was counted once. We followed the `dependencies` field, excluding development dependencies. These graphs reconstruct possible release-time resolutions.

We measured package archives with `pygount`, distinguishing internal size, external size (the aggregate code of resolved direct and transitive dependencies), and their sum, total size. The main analysis counts programming-language files. Two additional policies restrict measurement to JavaScript and TypeScript or extend it to templates and mixed-language files; all exclude JSON.

We computed net change, first–last slope, linear-regression slope, and growth fraction: the share of observed time with a positive derivative in a fitted quadratic. Growth requires a positive change or slope, or a growth fraction above 0.5; flat histories are stable. Net change and first–last slope yield identical direction classifications.

BENEVOL 2026, November 26–27, 2026, Madrid, Spain

© 2026 This work is licensed under a "CC BY 4.0" license.

3. Main findings

Table 1 summarizes growth across the complete sample under the main code policy. Total size grows in a majority of packages, with a median net increase of 139 LOC. However, increasing dependency counts are not the dominant pattern: only 36.9%–39.9% of packages show growth in the total number of dependencies.

Table 1
Packages classified as growing (% of all 2,091 packages). The first–last slope has the same classification as net change.

Measure	Net change	Linear slope	Growth fraction
Total code size	64.2	60.4	57.5
Internal code size	66.2	65.8	63.9
External code size	44.0	44.0	41.5
All dependency count	38.3	39.9	36.9

Internal-code growth is more widespread (63.9%–66.2%) than external-code growth (41.5%–44.0%). This is a statement about the prevalence of growth across packages, not the relative volume of code added. Indeed, dependencies supply more than half of total code in 52.7% of the full sample at the latest analyzed release.

The pattern changes among the 704 packages whose latest footprint is at least 90% external code. Their median external net growth is 2,590 LOC, compared with 6 LOC internally, and 70.0% show positive external net growth. Conversely, packages can grow without dependencies: among 408 packages that remain dependency-free throughout their analyzed histories, 72.3% show positive internal net growth.

4. Interpretation and discussion

The results are consistent with continuing growth in popular npm packages, but LOC does not directly measure functionality or user satisfaction. Archive contents can include generated or support code, and reconstructed dependency graphs do not capture every installation-specific choice. Popularity and history filters also limit generalization to other npm packages and ecosystems. Subgroup differences are descriptive associations, not evidence of causation.

Internal code, external code, and dependency counts capture different aspects of evolution. We seek feedback on interpreting continuing growth in independently maintained components and on replication in other ecosystems.

Full manuscript. This presentation abstract summarizes our manuscript of the same title,submitted to the *Journal of Systems and Software*. The full manuscript is available at: https://gsyc.urjc.es/jgb/tmp/Does_Software_still_Grow.pdf.

References

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

[2] A. Decan, T. Mens, P. Grosjean, An empirical comparison of dependency network evolution in seven software packaging ecosystems, Empirical Software Engineering 24 (2019) 381–416. doi:10.1007/s10664-017-9589-y.