Dependency trees did not grow: 25 of 39 frozen manifests held flat or shrank

42 application stacks resolved at 25 monthly snapshots across npm, PyPI and Maven Central, from registry metadata with nothing installed

Dmitriy Semenkevich · ORCID 0009-0009-3013-4978

18 August 2026 · dimhold.by

ABSTRACT

Every article about dependencies opens with the claim that trees keep inflating. This study resolves the claim instead of repeating it. 42 application stacks, 14 in each of 3 ecosystems, were resolved from registry metadata at 25 monthly snapshots between August 2024 and August 2026, with the manifest held frozen the whole time, for 1,050 resolutions and no installs. Of the 39 stacks that had a fully resolved baseline, 14 grew while 25 held flat or shrank. 28 of those 39 resolve cleanly from the first snapshot and carry the full 2 years; the other 11 have a later baseline, as late as January 2026. The table prints the base date for every row. On npm the count of accounts able to publish into a tree fell faster than the package count: webpack went from 78 packages and 35 publishing accounts to 68 and 26 without anyone editing a manifest. 3 further results come out of the same rows. Package age rises steeply with depth on npm and Maven Central, from a median of 36 days at depth 1 to 1,846 days at depth 5 on npm, while PyPI shows no such gradient. The same job costs 259 packages on npm and 5 on PyPI. Install time code execution, which exists on npm alone, reaches 11 distinct publishing accounts for a single test runner. Counts were checked against Google's deps.dev, which returned a graph for 29 of the 33 single root stacks; 14 of those matched exactly and 23 landed within 2 packages.

1. The question

A manifest is a short file. The tree it resolves to is not. The gap between the file and the tree is where every dependency argument happens. The argument is usually conducted with one number, the package count on the day somebody looked. The conclusion is usually that the number keeps rising.

That framing hides the question worth asking. A manifest that nobody edits still resolves to a different tree every month, because the ranges in it are ranges. So: **how much does a**

dependency tree change while the file describing it does not?

Answering that needs resolution as of past dates, which an install cannot give you. It also needs more than one ecosystem, because a claim about packaging that only holds on npm is a claim about npm.

2. Method

2.1 Nothing is installed

Every number here comes from the registries' own metadata: npm's package documents, PyPI's JSON API and Maven Central's POM files. A resolver walks the manifest, picks the version a fresh install would have picked on a given date and follows the runtime edges.

Installing measures one day and costs a `node_modules` directory per data point. Reading the registry resolves the same manifest as of any past date and cost 5,912 successful registry fetches across the 3 scans, which is what makes 25 snapshots per stack affordable rather than a sample of convenience. That total is the 3 scanners' own counters added up and it excludes the `deps.dev` comparison of section 2.4, which is a separate pass against a separate service.

The whole study reruns in under 3 minutes. The logs in the repository record the real runs of 18 August 2026: the npm scan fetched 611 package documents in 22 seconds, PyPI took 1,163 requests in 46 seconds, Maven Central took 4,138 requests in 103 seconds. All 3 scans finished with 0 errors. The counters and the timestamps behind those figures are in `logs/state.json`, `logs/state-pypi.json` and `logs/state-maven.json`.

2.2 The grid

14 stacks in each of 3 ecosystems, each resolved at 25 monthly snapshots from 2024-08-01 to 2026-08-18. That is 1,050 resolutions, one row of JSON each, in `results/npm.ndjson`, `results/pypi.ndjson` and `results/maven.ndjson`.

The stacks are ordinary applications rather than interesting ones: a web framework, an ORM with its driver, a test runner, an http client, a bundler, a linter and so on, chosen so

that the same job appears in all 3 ecosystems.

A snapshot's manifest is identical at every date. Only the registry moves.

2.3 What counts as a package

This is the logical graph: every distinct `name@version` reachable from the manifest through runtime edges. It is not the on disk tree, which installers hoist and deduplicate, changing the folder layout and the install size but not the set of code you end up trusting.

Runtime dependencies only. npm `devDependencies` and `optionalDependencies` are not walked, peer dependencies are not resolved, PyPI extras are never followed and Maven `test` and `provided` scopes are excluded. That is why `prisma`, `angular`, `next` and `quarkus` look small here: their weight arrives bundled, as peers or through a BOM.

Every root package counts as a node in its own tree. The `express` stack has 3 roots, `express` with `cors` and `body-parser`, so "`express` resolves to 71 packages" means 68 dependencies plus those 3. That convention is stated here because the next section is what forced it into the open.

2.4 Checked against an independent resolver

The counts were compared with Google's Open Source Insights (`deps.dev`), which publishes already resolved dependency graphs. The comparison ran on 19 August 2026 and its raw output is `validation.json` in the repository.

The first run came back +1 high almost everywhere: `vue` 23 against 22, `svelte` 20 against 19, `webpack` 68 against 67, `django` 3 against 2, `okhttp` 5 against 4. 15 of the 42 stacks were off by exactly 1, which is a definition rather than noise. `VALIDATION.md` names 10 of those 15. We count the root as a node and `deps.dev` reports the graph without it. The comparison now subtracts the roots and keeps `rawDelta` beside `delta` so the gap stays visible instead of being tidied away.

33 of the 42 stacks have a single root, which is the condition for a like for like diff. `deps.dev` returned no graph at all for 4 of them, `fastify`, `boto3`, `micronaut` and `guava`, so those 4 are not comparisons and are not counted as agreements. Of the 29 that

remain, 14 matched exactly and 23 landed within 2 packages. The 6 that did not are named in section 5.

Those figures are recomputed here from the shipped `validation.json` and they match the prose in the repository's `VALIDATION.md`, which reports the same 29, 14 and 23.

2.5 Conventions that decide the printed numbers

3 of them, stated because each one can be read the other way.

"Median" here is the upper of the 2 middle values. `report.mjs` takes `sorted[floor(n/2)]`, the nearest rank convention, rather than averaging the middle pair. It matters where a depth is reached by an even number of stacks: depth 5 on npm has 8 values and this convention prints 1,846 days where the averaged median is 1,386. Every median in section 3.1 is the nearest rank one, so the table and `node report.mjs` agree exactly.

Bytes are decimal. The install pilot in section 3.4 divides by 10 to the 6th, so 311 MB is 311,468,530 bytes rather than a power of 2 quantity.

A stack's base point is its first snapshot that resolved with nothing unresolved, not the first date on the grid. 28 of the 39 stacks reach that on 2024-08-01. The rest reach it later and the table in section 3 prints the base date per row rather than a single window in the heading.

3. Results

Resolved month by month with the manifest frozen, 25 of the 39 stacks with a usable baseline held flat or shrank. 13 of those held exactly flat, 12 shrank and 14 grew. On npm the number of accounts able to push code into a build fell faster than the package count.

3 stacks are excluded from that tally because no snapshot in their window resolved cleanly, which section 5 explains.

Every row ends at 2026-08-18. The base column is the stack's first cleanly resolved snapshot, so a row that starts in 2025 is a shorter observation than a row that starts in 2024 and the heading does not paper over the difference.

STACK	BASE	PACKAGES THEN TO NOW	PUBLISHING ACCOUNTS
`express` + `cors` + `body-parser`	2024-10-01	80 to **71**	20 to **16**
`webpack`	2024-08-01	78 to **68**	35 to **26**
`eslint`	2024-08-01	89 to **86**	35 to **33**
`fastify`	2024-10-01	56 to **49**	21 to **18**
`vue`	2024-10-01	24 to **23**	11 to **7**
`spring-boot-starter-data-jpa`	2024-08-01	65 to **54**	not published
`spring-boot-starter-security`	2024-08-01	35 to **28**	not published
`okhttp`	2024-08-01	12 to **5**	not published
`axios`	2024-08-01	9 to **30**	7 to 10
`micronaut-http-server-netty`	2024-08-01	42 to **66**	not published
`nestjs`	2025-02-01	12 to **21**	10 to 13

The ones that grew are named rather than hidden. `axios` more than tripled its tree and `micronaut` gained 24 packages. `nestjs` had the shortest run of the 11 rows, 18 months

rather than 2 years. `next` at 7.5 months is the shortest run in the study.

The second column is the one nobody prints. `webpack` lost 9 publishing accounts over 2 years while its manifest sat untouched. `vue` lost 4 of its 11. A lockfile freezes both columns, which is the practical reading of this table rather than a limitation of the method.

3.1 The deeper the package, the older the code

Median days since a package last published, grouped by its depth in the resolved tree, taken over the 14 stacks of each ecosystem. The count of stacks that reach each depth is in brackets. These are nearest rank medians, the upper of the 2 middle values where the count is even, as section 2.5 sets out.

DEPTH	NPM	PYPI	MAVEN CENTRAL
1, what you typed	36 (14)	77 (14)	**17** (14)
2	468 (13)	**48** (14)	70 (13)
3	671 (12)	622 (9)	341 (9)
4	1,251 (10)	**92** (6)	308 (7)
5	**1,846** (8)	103 (3)	1,329 (6)
8	4,057 (2)	not reached	**6,855** (1)

npm and Maven Central show the same shape. You install something released last month and 4 levels down you are running code last touched 3 to 5 years ago. On Maven Central the tail is worse: depth 8 sits just short of 19 years.

PyPI does not do this. Its deep packages are about as fresh as its shallow ones. Its depth 4 median is lower than its depth 1 median. Whatever Python packaging gets criticised for, its transitive layer is maintained.

3.2 The same job, 2 orders of magnitude apart

JOB	NPM	PYPI	MAVEN CENTRAL
web framework	71	**8**	37
ORM plus driver	79	**4**	54
test runner	**259**	5	9
http client	30	7	5

`jest` resolves to 259 packages. `pytest` does the same job in 5. That is a factor of 52 for one job. It is a packaging culture rather than a language.

The depth figure follows the count. 98% of the `jest` tree sits at depth 3 or deeper, which is to say no dependency you named refers to it.

3.3 Install time code and who owns it

A package can execute its own code during installation through `preinstall`, `install`, `postinstall` or `prepare`. Counting those across the resolved tree and counting the distinct npm accounts that published them:

STACK	PACKAGES	INSTALL SCRIPTS	DISTINCT ACCOUNTS
`jest`	259	**14**	**11**
`eslint`	86	10	6
`typeorm` + `pg`	79	8	6
`webpack`	68	6	4
`express` + `cors` + `body-parser`	71	2	1

Across the 14 npm stacks there are 49 packages carrying an install hook. Only 3 of those 49 sit at depth 1. Installing one test runner runs install time code belonging to 11 separate accounts, none of them shallower than depth 3.

The row that matters most is the one that cannot be drawn. **On Maven Central this number does not exist**, because resolving a dependency never executes its code. On PyPI it is bounded: across these 14 stacks, 0 packages resolved to a source only distribution, which is the case in which a `setup.py` would run at install time.

3.4 The count explains the least

From the one run that did touch a disk, in `logs/npm-scan-install-pilot.log`. The stacks here are not the stacks of the table above: this pilot installs one root package each, so its `express` row is `express@5` alone with 66 directories on disk, while the 71 in section 3 is the logical graph of `express` with `cors` and `body-parser`. The 2 numbers count different things about different manifests.

STACK	PACKAGES INSTALLED	BYTES ON DISK
`next`	21	**311 MB**
`nestjs`	21	6.5 MB
`express`	66	2.2 MB
`@prisma/client`	1	**77 MB**

`next` and `nestjs` resolve to the same number of packages and differ by a factor of 48 in weight. `@prisma/client` is a single package heavier than a 66 package tree by a factor of 35. The metric everybody quotes is the one that tracks the disk worst, because bundled code and binaries are invisible to a package count.

4. What it means

The growth story is wrong for the stacks people actually run. It is wrong in an interesting direction. 12 of the 39 stacks ended smaller than they started. The `express` stack lost 9 packages and `webpack` lost 10, with no manifest edit on either side. Whatever the registry is doing, it is not only adding.

That last sentence is an npm and Maven Central sentence. 11 of the 12 shrinking stacks are in those 2 ecosystems: npm shrank 5 times and grew 4, Maven Central shrank 6 and grew 3. PyPI shrank exactly once, `pandas` going from 6 packages to 5, against 7 stacks that grew. On PyPI the growth story is closer to right than wrong and the headline of this paper does not hold there.

But the useful reading is not the package count at all. It is the second column. The set of humans who can publish code into your build changes without your manifest changing. On the 5 npm stacks that shrank it changed faster than the package count did every time: `express` lost 11% of its packages and 20% of its accounts, `webpack` 13% against 26%, `vue` 4% against 36%, `fastify` 13% against 14%, `eslint` 3% against 6%. A tree that shrinks by 10 packages is not automatically safer, though a tree that loses 9 publishing accounts probably is. Neither fact is visible from the file you edit.

The 2 columns almost never point in opposite directions. Across the npm stacks where both moved, 5 went down together and 2 went up together. `svelte` is the only stack in the study where they part, gaining 2 packages while losing a publishing account.

Depth is where the other 3 results converge. Age climbs with depth on 2 of the 3 ecosystems. 46 of the 49 npm install hooks sit at depth 2 or deeper. Most of a large tree is unreachable from anything you named. The layer you chose is maintained. The layer under it is where the old code and the install scripts both live.

That PyPI breaks the depth pattern is the result that says the pattern is not a law of package management. It is a property of an ecosystem. One of the 3 here does not have it.

5. Threats to validity

14 stacks per ecosystem, one registry each, one day of measurement. This is a comparison rather than a benchmark. No significance test is claimed anywhere in it.

The Maven resolver is incomplete on purpose. It implements parent POMs, property interpolation, `dependencyManagement` and imported BOMs, together with scope and optional filtering. It does not implement profiles, exclusions or relocation, each of which would shrink a real tree slightly. Maven counts here are an upper bound.

PyPI markers are evaluated for Python 3.12 on Linux. A different interpreter or platform resolves a different tree.

3 stacks are missing from the drift tally. `jest`, `typeorm` and `netty` never produced a snapshot with 0 unresolved packages in their window, so no honest baseline exists for them and they are excluded rather than filled in. Their present day counts still appear elsewhere, carrying 4 unresolved packages in the case of `jest` and 1 each for `typeorm` and `netty`.

6 of the 29 comparable stacks disagree with `deps.dev` by more than 2 packages. Counting instead over all 38 rows for which `deps.dev` returned a graph, single root or not, there are 11. The largest of the 5 extra ones is `typeorm` at 79 here against 27 there. Those 5 are multi root stacks, where our count covers several manifest lines and theirs covers 1, so the gap is a frame difference rather than a resolver difference and it is not compared above. The disagreements below are not all understood. `log4j-core` resolves to 2 packages here against 14 there, because its POM marks 15 dependencies optional and we skip optional edges. `jest` resolves to 259 here against 323 there, consistent with optional and platform specific edges we drop. `httpx` and `vertx-web` are the same boundary in the other direction. `eslint` at 86 against 68 is not explained. `netty-all` **is the one to distrust**: `deps.dev` reports an empty graph while the POM we read lists 338 dependency entries with no optional flags. That 338 is prose in `VALIDATION.md` written while the POM was open; the shipped `validation.json` carries only the 2 counts, 143 against 0, so a reader who wants the 338 has to open the POM rather than the artifact. No claim in this paper rests on `netty`. None should until that is understood.

The age by depth medians hide a wide spread. Depth 5 on npm has a median of 1,846 days across 8 stacks whose individual values run from 85 days to 2,888. The gradient is a property of the middle of the distribution, not a promise about any one tree.

The publisher column exists for npm alone. PyPI publishes no per version publisher account and Maven Central publishes only prose in `<developers>`. That absence is a result about the registries rather than a gap in the code, but it means the claim about publishing accounts is an npm claim.

Install hooks are counted from the version's `scripts` block. A declared hook is not proof that meaningful code runs. `prepare` does not run for a consumer installing from the registry, so 49 is an upper bound on the 14 stacks.

6. Prior work

Every neighbouring question here has an owner. 2 of them sit closer than the repository's own notes recorded when it was published.

2 companion measurements sit on the same registry and are quoted here because their numbers disagree with these. A census of every npm package (10.5281/zenodo.22128843) finds 84.9% of the registry listing exactly 1 maintainer and 65.8% silent for 2 years. A walk of installed `node_modules` trees (10.5281/zenodo.22128826) finds 50.0% and 44.7% over the 1,598 names those trees actually contain. The registry is a warehouse and most of it is a single publish that nobody ever installed, so any share taken over all of npm reads far more abandoned than the same share taken over what a project resolves to. This study sits with the second: it resolves manifests, so its stacks are drawn from the part of the registry people use.

Cross ecosystem comparison of dependency networks is a settled genre. Decan, Mens and Grosjean compared the evolution of 7 packaging ecosystems from Libraries.io data (DOI, Empirical Software Engineering 2019). `deps.dev` publishes resolved graphs for 6 ecosystems and Ecosyste.ms maintains continuously updated registry metadata.

Transitive counts across many ecosystems are taken, taken this year. "How Deep Does Your Dependency Tree Go? An Empirical Study of Dependency Amplification Across 10 Package Ecosystems" (arXiv:2512.14739, December 2025) measures the ratio of transitive to direct dependencies over 500 projects in 10 ecosystems and reports mean amplification of 24.70 on Maven against 4.48 on Go Modules. That is the same shape as section 3.2 on a far larger sample. What it does not do is resolve one frozen manifest repeatedly over time, which is the axis this study adds.

Historical resolution is taken for npm. The `npm-follower` dataset archives every npm version as published (DOI, ESEC/FSE 2023). The same group built a time travelling

npm resolver to study how semver updates flow (arXiv:2304.00394, April 2023). Their question is the propagation of updates rather than the drift of a frozen tree, but resolving npm manifests as of past dates is already done. No equivalent was found for PyPI or Maven Central.

Trust through the tree is measured for npm. Zimmermann and colleagues counted the packages and maintainers an install implicitly trusts (USENIX Security 2019). Zahan and colleagues flagged install scripts and maintainer reach as weak link signals across 1,630,000 npm packages (arXiv:2112.10165, December 2021). Duan and colleagues compared install time attack surface across npm, PyPI and RubyGems (NDSS 2021).

Publishing authority now has its own measurement. "On Good Authority: Release Authority Measurement for Registry Mediated Package Ecosystems" (arXiv:2606.22593, 2026) tracks how the publisher account, repository link, workflow and signing evidence behind a release change from one version to the next. That is the same underlying object as the publishing account column in section 3, approached per release rather than per tree.

Outdatedness has a literature under the name technical lag. Decan, Mens and Constantinou measured it for npm (DOI, ICSME 2018), Kula and colleagues asked whether developers update at all (DOI, Empirical Software Engineering 2017). Soto-Valero and colleagues measured bloated dependencies in Maven (DOI, Empirical Software Engineering 2021).

What was not found is package age broken down by depth in the resolved tree and compared across ecosystems, which is section 3.1. Nor was the same frozen manifest resolved month by month in 3 ecosystems at once, which is section 3.

Searched arXiv, Semantic Scholar and GitHub on 29 August 2026. Semantic Scholar rate limited most queries and general web search was unavailable that day, so the open web outside those sources is not claimed as checked.

The novelty check for this measurement was written on 27 August 2026, 9 days after the run of 18 August, so it stands after the counting rather than before it. That order is wrong and is recorded as such; from 27 August 2026 the check is required beside the disproof condition, before the first count.

7. Availability

The scanners, the resolver for each ecosystem, the raw rows for all 1,050 resolutions, the unedited run logs and the `deps.dev` comparison are in the repository. `node report.mjs` rebuilds every table above from the raw rows. It was rerun from a clean copy of the published repository while this paper was written: it reproduces the shipped `RESULTS.txt` exactly. The archived release carries the DOI above.

One half of section 2.4 does not reproduce offline. `validation.json` records what `deps.dev` answered, not the answers themselves: the response cache the comparison was built from is not in the repository, so rerunning `validate-depsdev.mjs` calls the live service and compares against whatever it returns today rather than against 19 August 2026. Our side of every row recomputes from the shipped rows; their side is a claim about what a third party said on a date.

8. References

- Alexandre Decan, Tom Mens, Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering, 2019. DOI 10.1007/s10664-017-9589-y. https://doi.org/10.1007/s10664-017-9589-y
- Alexandre Decan, Tom Mens, Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. ICSME, 2018. DOI 10.1109/ICSME.2018.00050. https://doi.org/10.1109/ICSME.2018.00050
- Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, Katsuro Inoue. Do developers update their library dependencies? Empirical Software Engineering, 2017. DOI 10.1007/s10664-017-9521-5. https://doi.org/10.1007/s10664-017-9521-5
- César Soto-Valero, Nicolas Harrand, Martin Monperrus, Benoit Baudry. A comprehensive study of bloated dependencies in the Maven ecosystem. Empirical Software Engineering, 2021. DOI 10.1007/s10664-020-09914-8. https://doi.org/10.1007/s10664-020-09914-8
- Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, Michael Pradel. Small world with high risks: a study of security threats in the npm ecosystem. USENIX

Security, 2019. arXiv:1902.09217. https://arxiv.org/abs/1902.09217

- Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, Laurie Williams. What are weak links in the npm supply chain? arXiv, 2021. arXiv:2112.10165. https://arxiv.org/abs/2112.10165
- Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. NDSS, 2021. https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/
- Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell. `npm-follower`: a complete dataset tracking the npm ecosystem. ESEC/FSE, 2023. DOI 10.1145/3611643.3613094. https://doi.org/10.1145/3611643.3613094
- Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell. A large scale analysis of semantic versioning in npm. arXiv, 2023. arXiv:2304.00394. https://arxiv.org/abs/2304.00394
- Jahidul Arafat. How deep does your dependency tree go? An empirical study of dependency amplification across 10 package ecosystems. arXiv, 2025. arXiv:2512.14739. https://arxiv.org/abs/2512.14739
- Igor Santos-Grueiro. On good authority: release authority measurement for registry mediated package ecosystems. arXiv, 2026. arXiv:2606.22593. https://arxiv.org/abs/2606.22593
- Dmitriy Semenkevich. A census of 4,296,340 npm packages. 2026. DOI 10.5281/zenodo.22128843. https://doi.org/10.5281/zenodo.22128843
- Dmitriy Semenkevich. Duplicate copies hold 14.0 to 17.2% of node_modules. 2026. DOI 10.5281/zenodo.22128826. https://doi.org/10.5281/zenodo.22128826
- Google Open Source Insights. deps.dev. https://deps.dev/
- Ecosyste.ms. Open source ecosystem metadata. https://ecosyste.ms/

CITE THIS

Semenkevich, D. (2026). Dependency trees did not grow: 25 of 39 frozen manifests held flat or shrank. dimhold.by. https://doi.org/10.5281/zenodo.22128854

software supply chain · dependency resolution · registry metadata · npm · PyPI · Maven Central · technical lag · empirical software engineering