Duplicate copies hold 14.0 to 17.2% of node_modules and most of them are the same version twice

5 real projects walked on disk, 5,127 installed packages, duplicates charged in bytes rather than counted as graph nodes

Dmitriy Semenkevich · ORCID 0009-0009-3013-4978

27 August 2026 · dimhold.by

ABSTRACT

Published studies of dependency bloat resolve a package graph from registry manifests and count distinct names. That unit cannot see a package that is installed twice. This study walks node_modules on disk in 5 real projects, 3 large Angular front ends belonging to one former client and 2 small tools of the author and counts physical copies, bytes and files. 5,127 packages were found on disk behind 192 lines of dependencies. On the 3 front ends, copies after the first hold 14.0%, 14.4% and 17.2% of all bytes under node_modules, which is 102.5 MiB, 52.2 MiB and 60.6 MiB. Byte figures are binary throughout because the scanner divides by 1024. Between 58.8% and 64.1% of those copies are the same name at the same version rather than a version conflict a resolver had to keep apart. Counting copies and weighing them disagree by about a factor of 2: copies are 30.8% to 31.6% of installed packages but only 14.0% to 17.2% of bytes, because a duplicate copy is roughly half the size of an average installed package. Bytes are charged only to copies after the largest one, so every figure is the conservative reading of extra. The 2 small projects carry almost no duplication. A sample of 5 projects gives no base rate for anything and none is claimed.

1. The question

A project lists a few dozen names under `dependencies`, runs `npm install` and gets a directory. How much arrives per line typed is a published quantity. It has a name, the amplification factor. This study does not add to it.

What is missing is the unit. A resolved dependency graph has one node per package name. A disk has one directory per copy and npm writes a second copy whenever a nested requirement cannot be hoisted to the top. Those copies are foreign code sitting in the tree,

they are read by tooling, they are scanned by editors, they are shipped in container images. A graph counted by name cannot see any of them.

So the question here is narrow and physical. Walk `node_modules`, count what is actually lying in it and charge the extra copies in bytes.

2. Method

2.1 The sample

5 projects, each with a `package.json` and an installed `node_modules` beside it. A project without an installed tree is not in the sample, because counting from a lock file would be counting something else.

PROJECT	KIND	DIRECT DEPENDENCIES
`frontend-a`	Angular front end with Selenium end to end tests	70
`frontend-b`	Angular front end with Selenium end to end tests	51
`frontend-c`	Angular front end with Selenium end to end tests	62
`fault-mcp`	MCP server library	6
`social-media`	TypeScript CLI	3

The 3 front ends belong to a former client. They are published as numbers only, with no names and no paths. The 2 small projects belong to the author and are a deliberate contrast: without something small in the sample the large numbers have nothing to sit against.

`direct` is the count of names under `dependencies` plus `devDependencies` in the root manifest. The 5 projects together declare 192 of them.

2.2 What counts as installed

Every directory under `node_modules` holding a `package.json`, nested `node_modules` included. Dot directories are skipped, so `.bin` and `.package-lock.json` are not packages. A scoped package counts once and the `@scope` directory itself does not count.

Duplicate versions are deliberately not collapsed. A library present at 3 versions is 3 copies of foreign code in the tree and the tree is the subject. The collapsed figure is reported beside it as `distinctNames`, so the difference stays visible instead of being a choice made silently.

Bytes and file counts for a package exclude its own nested `node_modules`, so nothing is counted twice. Symbolic links are not followed. Tree depth comes from `npm ls --all --json`, which exits non zero on an inconsistent tree while still emitting usable JSON on stdout; that case is recorded as degraded and the run continues, because old projects are inconsistent and that is the field rather than an error.

Byte figures are binary. The scanner divides by 1024 twice, so every quantity the data calls MB is a MiB and this paper writes MiB and KiB throughout. The 363.3 MiB below is 380,920,515 bytes, which a decimal reading would print as 380.9 MB.

Most of this is fixed in `CRITERIA.md`, committed before the first number existed, with a disproof condition written at the same time: if the ratio had come out around 3 to 5 and been the same for large projects and small ones, there would have been nothing to publish.

2 things are not in that file. It defines `installed`, `distinctNames`, `bytes` and `files` and it says duplicate versions are not collapsed. It does not define a duplicate copy and it does not contain the charging rule of the next section. Both were settled while the counting was being written, so the headline of this paper rests on a definition the criteria file does not hold. The disproof condition that file does hold is about the ratio in section 3.3 rather than about duplicates.

2.3 How a duplicate is charged

Group every installed copy by package name. For a name with more than one copy, sort the copies by size and charge everything after the largest one.

2 consequences follow and both are on purpose. The largest copy is free, so the figure is the smallest defensible reading of extra. Copies of the same name at the same version are charged like any other, because identical bytes written twice occupy the disk twice.

2.4 The registry pass

One request per distinct package name, over full packuments. The abbreviated packument format is cheaper and carries neither `maintainers` nor `time`; a first pass used it and returned `maintainers: 0` for all 1,598 names. That result was impossible on its face, which is the only reason it was caught. The pass was rerun.

Maintainer counts and publish dates come from the registry by name, never from the installed manifest, because an installed manifest is not required to carry a `maintainers` field. The disk walk and the join finished on 27 August 2026, ages are measured against a snapshot date of 2026-08-26 and the registry data was pulled on that day.

3. Results

On the 3 front ends, copies after the first hold 14.0% to 17.2% of every byte under `node_modules` and 58.8% to 64.1% of those copies are the same name at the same version.

PROJECT	INSTALLED	DISTINCT NAMES	COPIES AFTER THE FIRST	DUPLICATE MIB	SHARE OF BYTES	SAME VERSION
`frontend-a`	1,729	1,197	532	52.2	**14.4%**	313
`frontend-b`	1,542	1,055	487	102.5	14.0%	312
`frontend-c`	1,709	1,181	528	60.6	**17.2%**	332
`fault-mcp`	141	139	2	0.03	0.0%	1
`social-media`	6	6	0	0	0.0%	0

On `frontend-a` that is 532 copies holding 7,904 files inside a tree of 1,729 packages, 363.3 MiB and 50,649 files. Those last 2 figures are the walk of the whole directory and they come from `out/projects.json` rather than from the per package file: adding up the 1,729 package rows gives 362.33 MiB and 50,306 files, the remainder being files that sit under `node_modules` outside any package directory. 165 names are present at more than one version at once. The equivalent counts for the other 2 front ends are 137 and 154 names.

3.1 Counting copies overstates the disk cost by about a factor of 2

The 2 ways of reading the same duplicates disagree and the disagreement is stable across all 3 front ends.

PROJECT	COPIES AS A SHARE OF INSTALLED PACKAGES	COPIES AS A SHARE OF BYTES	RATIO
`frontend-a`	30.8%	14.4%	2.14
`frontend-b`	31.6%	14.0%	2.26
`frontend-c`	30.9%	17.2%	1.80

The cause is size. A duplicated copy averages 100.6 KiB, 215.6 KiB and 117.5 KiB on the 3 front ends against an average installed package of 214.6 KiB, 470.7 KiB and 210.6 KiB. Duplication concentrates in small packages, so a headline built on the count of copies claims roughly twice the disk it can support.

The 2 quantities in that sentence do not share a denominator. The byte share divides duplicate bytes by the whole tree walk. The average package size divides the sum of the per package rows by the package count. On `frontend-a` and `frontend-c` the 2 totals are 0.26% apart and nothing turns on the choice. On `frontend-b` they are 3.39% apart, so its duplicate share is 14.0% against the tree and 14.47% against the sum over packages. Every share in this paper uses the tree, because the tree is what the disk actually holds.

3.2 Most duplication is not version resolution

313 of `frontend-a`'s 532 duplicate copies, 58.8%, are the same name at the same version. On `frontend-b` it is 312 of 487, 64.1% and on `frontend-c` 332 of 528, 62.9%.

Those are not incompatible versions that a resolver was forced to keep apart. They are identical packages written to disk more than once, because the hoisting rule that decides where a package lands is positional rather than content aware.

3.3 The ratio does not track project size

PROJECT	DIRECT	INSTALLED	INSTALLED PER DIRECT
`frontend-b`	51	1,542	30.2
`frontend-c`	62	1,709	27.6
`frontend-a`	70	1,729	24.7
`fault-mcp`	6	141	23.5
`social-media`	3	6	2.0

A 6 dependency MCP server library sits in the same band as 3 large front ends. The outlier is the 3 dependency CLI, whose dependencies happen to be `typescript`, `tsx` and `@types/node`. What gets picked decides the ratio and project size does not, though 5 projects cannot establish more than that.

3.4 Who is behind the names

Across the union of all 5 projects, 1,598 distinct names arrive from 852 distinct publishing accounts. 799 of the names, 50.0%, list exactly one maintainer. 930 names, 58.2%, had no publish in over a year at the snapshot date and 715, 44.7%, none in over 2 years.

ACCOUNT	NAMES	SHARE OF 1,598
`ljharb`	141	8.8%
`nicolo-ribaudo`	132	8.3%
`hzoo`	128	8.0%
`existentialism`	128	8.0%
`jlhwung`	127	7.9%
`sindresorhus`	118	7.4%
`jonschlinkert`	101	6.3%

4 of the top 5 are Babel maintainers, so those rows overlap heavily by construction: a monorepo publishes many names under one set of accounts. `ljharb` and `sindresorhus` do not have that explanation.

4. A resolved graph counts names, a disk holds copies

The gap between 1,197 distinct names and 1,729 installed packages on `frontend-a` is the whole finding. Everything published about dependency amplification lives on the left of that gap. The 532 packages on the right are invisible to it by construction and they are just under a third of what is on the disk.

3 practical readings follow, in decreasing order of confidence.

Deduplication has a real ceiling and the copy count overstates it. On these front ends a perfect content addressed store would recover 14.0% to 17.2% of the tree, not the 31% that counting copies implies.

Most of what it would recover is not a resolution conflict. Around 60% of the copies are byte identical duplicates of the same version, which is the part a store keyed by content removes without deciding anything.

Anything that reads the tree pays for the copies. The count is what an editor indexes, what a scanner walks and what a naive image layer ships. There the 31% figure is the relevant one rather than the byte share. The 2 readings answer different questions and neither replaces the other.

5. Threats to validity

5 projects, 3 of which are near siblings. They are Angular front ends of one former client, built by one team on one toolchain and their duplicate shares cluster in a narrow band partly for that reason. Nothing here is a base rate for JavaScript projects and a wider sample is a separate measurement that this one does not stand in for.

Installed does not mean executed. `node_modules` carries tests, documentation and build artefacts of other people’s packages. That is part of what arrives on the machine

and it is not an answer to what runs. Separating live code from dead code needs a different instrument.

This is not a vulnerability count. `npm audit` counts against its own database with its own method and mixing the 2 would produce a number belonging to neither.

Hard links are not detected. The walk sums file sizes, so a package manager that stores content once and links it into place would still be charged for every copy here. All 5 projects were installed with npm, which does not do that, but the instrument would misreport pnpm and the numbers should not be carried over to it.

On disk allocation is not measured either. File sizes are summed, not blocks, so the true disk cost of many small files is higher than the figures above rather than lower.

Depth on `frontend-b` comes from an `npm ls` run that exited non zero on an inconsistent tree and is flagged as degraded in `results.json`. It is reported because the tree is the field as it was found.

6. Prior work

The framing this study opens with is published and it is named here so that nothing in it reads as a claim of priority.

- How Deep Does Your Dependency Tree Go? An Empirical Study of Dependency Amplification Across 10 Package Ecosystems (arXiv:2512.14739, December 2025) defines dependency amplification as the ratio of transitive to direct dependencies and reports a mean of 4.32 for npm, with 12% of projects above 10. The ratio in section 3.3 is a larger and different quantity because it counts every directory holding a `package.json`.
- A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem (arXiv:2001.07808, January 2020) analyses 9,639 Java artifacts and 723,444 dependency relationships and finds 75.1% of the relationships bloated. The unit is a dependency relationship in a build, not a copy on a disk.
- Detecting and removing bloated dependencies in CommonJS packages (arXiv:2405.17939, May 2024) is the closest work in this ecosystem. It traces file system

access at runtime over 91 CommonJS packages and 50,488 dependencies and finds 50.6% of them never accessed. It watches the file system, as this study does, but it asks which dependencies are used rather than how many times each one is present.
- Dependency Managers Don't Manage Your Dependencies makes the qualitative case for the same discomfort without measuring it.

2 companion measurements of ours look at the same ecosystem from the registry side and their figures for the same 2 quantities are far higher. A census of all 4,296,340 npm packages (10.5281/zenodo.22128843) finds 84.9% listing exactly 1 maintainer and 65.8% silent for 2 years, against the 50.0% and 44.7% in section 3.4. A resolution of 42 frozen manifests from registry metadata (10.5281/zenodo.22128854) counts the same kind of tree as a logical graph rather than as directories. The gap is the frame: the census counts the registry as a warehouse, where most entries are a single publish nobody ever installed, while section 3.4 counts only names that a real project resolved to and put on a disk.

Duplication on disk is thoroughly occupied by tooling and not by measurement. `npm dedupe` exists as a command, pnpm stores package content once and links it, Yarn Plug and Play removes the directory layout altogether and a general web search on 27 August 2026 returned many practitioner posts on reclaiming space under `node_modules`. What was not found is a study that walks real installed trees and reports the duplicate share in bytes, the share of duplicates that are copies of one version and the gap between counting copies and weighing them.

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.

One order of work is recorded because it was wrong. The novelty check for this measurement was written on 27 August 2026, after the numbers existed. From that date the check is required beside the disproof condition, before the first count.

7. Availability

The 3 scripts, the per package byte and file counts for every installed copy, the registry pass output and the criteria file are in the repository. The archived release carries the DOI above.

Every duplicate figure in section 3 recomputes from the published per package data in `out/packages-*.json`, which carries a name, a version, a nesting depth, a byte count and a file count for all 5,127 installed copies. 2 figures in that section come from somewhere else and are marked where they appear. The 363.3 MiB and 50,649 files quoted for `frontend-a` are the whole tree walk in `out/projects.json`. Every byte share divides by that same walk.

The published `scan.mjs` cannot produce any of it. It writes `{name, version, nesting}` for each package, with no byte count and no file count, so it is not merely missing the duplicate accounting: it could not have written `out/packages-*.json` at all. No script in the repository produces the data the repository ships. The figures verify against the data and the data verifies against nothing published. The fix is one commit carrying the scanner that actually ran. It has not been made.

The registry aggregates in section 3.4 recompute from `out/maintainers.ndjson`, which does verify against the published `analyze.mjs`.

8. References

- 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
- César Soto-Valero, Nicolas Harrand, Martin Monperrus, Benoit Baudry. A comprehensive study of bloated dependencies in the Maven ecosystem. arXiv, 2020. arXiv:2001.07808. https://arxiv.org/abs/2001.07808
- Yuxin Liu, Deepika Tiwari, Cristian Bogdan, Benoit Baudry. Detecting and removing bloated dependencies in CommonJS packages. arXiv, 2024. arXiv:2405.17939. https://arxiv.org/abs/2405.17939

- Dependency managers do not manage your dependencies. cpojer.net. https://cpojer.net/posts/dependency-managers-dont-manage-your-dependencies
- 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. Dependency trees did not grow: 25 of 39 frozen manifests held flat or shrank. 2026. DOI 10.5281/zenodo.22128854. https://doi.org/10.5281/zenodo.22128854

CITE THIS

```
Semenkevich, D. (2026). Duplicate copies hold 14.0 to 17.2% of node_modules and most of them are the same version twice. dimhold.by. https://doi.org/10.5281/zenodo.22128826
```

npm · dependency bloat · node_modules · duplicate packages · software supply chain · empirical software engineering · on disk measurement