How Reproducible Are Build and Test Outcomes of Dependency Updates in JavaScript/npm Projects?

An Empirical Study of Dependabot npm Update Pull Requests

Vladyslav Maksymiuk[1]

Supervisor(s): Sebastian Proksch[1], Cathrine Paulsen[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

June 20, 2026

Name of the student: Vladyslav Maksymiuk
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Cathrine Paulsen, Johan Pouwelse

Abstract

Automated dependency-update tools such as Dependabot help projects keep dependencies secure and compatible, and CI results give maintainers practical evidence for deciding whether such updates can be merged, delayed, or require closer inspection. This evidence is useful only if the same update state tends to produce the same observable outcome when executed again. Prior work has studied reproducible dependency-update failures in ecosystems such as Java/Maven [5]; this thesis examines whether similar reproducible CI-like evidence can be obtained for JavaScript/npm projects, and to what extent install, build, and test outcomes are reproducible in an ecosystem where lockfiles, package-manager behavior, browser tooling, native addons, and peer dependencies may introduce instability.

Building on prior work on automated dependency updates and reproducible dependency-update benchmarks [4–6], this study starts from 3,083 Dependabot candidates and analyzes 2,777 npm update pull requests through Dockerized Node.js executions of install, build, and test phases, producing 3,142 experiments and 9,426 runs. The results show that most outcomes are reproducible, but 152 experiments (4.8%) and 111 PRs (4.0%) are non-reproducible. Instability is concentrated in tests and appears more often in browser-coupled, native-addon-risk, peer-dependency-risk, and Yarn-based projects. These findings suggest that CI remains useful evidence for npm dependency-update decisions, but practitioners and researchers should treat single-run outcomes as incomplete evidence in instability-prone settings.

1 Introduction

Modern software projects depend heavily on third-party packages. Keeping these packages up to date is necessary for security, bug fixes, and compatibility, but each update also changes part of the software supply chain. Prior studies of dependency freshness and library migration show that outdated dependencies are common and that security advisories do not automatically lead to prompt updates [1, 2]. Automated tools such as Dependabot reduce the manual effort of tracking releases by opening pull requests (PRs) that projects can evaluate through their usual continuous integration (CI) workflows [3, 4]. In this workflow, CI acts as practical evidence: if installation, build, and tests pass, the update appears suitable for further review or merge; if one of these stages fails, the update requires attention.

This paper studies a weakness in that evidence model. A CI result is informative only if the same source state tends to produce the same observable outcome when executed again. If an update PR passes once and fails later under a similar environment, then a single run is not a stable update-assessment signal. This risk is especially relevant for JavaScript/npm projects, where deep dependency graphs, flexible version ranges, generated lockfiles, browser-oriented tests, native Node modules, and multiple package managers can affect installation, build, and test execution.

Prior work shows that this problem is worth studying but does not directly answer it. Dependabot studies report rejection, delay, and manual review as common parts of automated dependency management [4, 6]. CI studies show that continuous integration is widely used in open-source projects and that build failures are a recurring maintenance signal [7, 8]. Work on test-based dependency-update assessment shows that project tests may not expose all dependency faults [9]. BUMP preserves breaking Java/Maven dependency updates as reproducible benchmark cases [5]. These studies motivate update evaluation, test evidence, and controlled execution, but they do not measure whether mined JavaScript/npm update PRs produce reproducible observable outcomes under repeated execution.

The main research question is therefore:

> **RQ:** *How reproducible are install, build, and test outcomes of dependency updates in JavaScript/npm projects?*

To answer this question, the study starts from a Dependabot PR dataset and public GitHub metadata, qualifies runnable JavaScript/npm projects, and executes install, build, and test phases repeatedly in Dockerized Node.js environments. The resulting run evidence is classified into reproducible success, reproducible failure, non-reproducibility, and insufficient evidence, and npm failure patterns are compared with the Java/Maven dependency-update failures reported in BUMP.

The key answers are:

- Most npm update outcomes are stable under repetition, but non-reproducibility is still detected: 111 of 2,777 analyzed PRs show unstable evidence.
- Instability is concentrated in test execution and appears more often in browser-coupled, native-addon-risk, peer-dependency-risk, and Yarn-based projects.
- The comparison with BUMP shows that broad failure categories can support cross-ecosystem comparison, but npm outcomes require finer-grained labels than the Java/Maven taxonomy provides.

The contributions are:

- a pipeline for mining, filtering, repeatedly executing, and classifying JavaScript/npm dependency-update PRs;
- an outcome-reproducibility model that distinguishes reproducible success, reproducible failure, non-reproducibility, and insufficient evidence;
- empirical evidence from 2,777 analyzed update PRs and 3,142 experiment analyses, including a comparison with the BUMP Java/Maven benchmark.

2 Related Work

This study builds on research showing that dependency-update pull requests are decision-support artifacts rather than automatically trusted fixes. Dependency-management studies establish why these updates matter:

outdated dependencies are common, security advisories do not always lead to prompt updates, and automated tools can increase update activity [1–3]. Dependabot-specific studies further show that maintainers still configure, inspect, delay, reject, and replace bot-generated updates [4, 6]. These results imply that maintainers interpret evidence attached to update PRs, especially build and test results. The question addressed in this paper is not whether such PRs are accepted, but whether the technical evidence used to judge them is reproducible.

Prior work on CI and test-based dependency assessment motivates this focus on evidence quality. CI is widely used in open-source projects, and build failures are a recurring maintenance signal [7, 8]. At the same time, project tests do not necessarily exercise all dependency usages, so a passing update run is not a proof of correctness [9]. This paper studies a complementary property: before asking whether tests are complete, it asks whether the same update state produces stable observable outcomes when executed repeatedly. A stable failure and an unstable failure are therefore treated as different empirical outcomes.

The repeated-execution design is grounded in reproducibility and flaky-test research. Reproducible-build work emphasizes controlling source state, dependencies, and execution environments as part of software supply-chain integrity [10]. Flaky-test studies show that unchanged code can pass and fail nondeterministically, including in JavaScript projects [13, 14], and that rerunning tests is a practical way to expose instability [15,16]. This study applies the same idea at the level of whole dependency-update experiments, where non-reproducibility may arise during installation, package resolution, native compilation, browser setup, external downloads, timeout behavior, or test execution.

The closest methodological comparison is BUMP, a benchmark of reproducible breaking dependency updates in Java/Maven projects [5]. BUMP motivates preserving source and dependency state and provides useful failure categories for cross-ecosystem comparison. However, BUMP starts from known breaking updates, whereas this paper measures whether mined npm update PRs produce reproducible evidence at all. The classification scheme therefore includes reproducible successes, reproducible failures, non-reproducible outcomes, and insufficient-evidence cases.

Finally, npm ecosystem studies explain why Java/Maven assumptions cannot be transferred directly. npm has dense dependency graphs, high reuse, many small packages, evolving dependency networks, and ecosystem-specific versioning and breaking-change behavior [17–23]. These findings motivate the npm-specific features used in this study, including package-manager choice, lockfiles, dependency graph surface, peer and optional dependency risk, lifecycle scripts, native-addon markers, browser-coupled tooling, package metadata, and selected Node.js environment.

3 Methodology

This study is an empirical software engineering study of outcome reproducibility in dependency-update pull requests. The measured object is not a byte-identical build artifact. Instead, the study asks whether repeated executions of the same repository state, source variant, selected Node.js environment, and install/build/test policy produce the same observable evidence: phase outcomes, terminal status, error type, and failure signature. The use of Dockerized execution follows prior work that treats containers as a practical mechanism for controlling software environments in reproducible computational and software-engineering research [11, 12].

The methodology follows a six-stage pipeline that moves from candidate update selection to repeated execution and final analysis. Figure 1 shows how these stages connect and how the resulting evidence flows into the research-question analyses.

S1 Input normalization. Candidate Dependabot pull requests and GitHub metadata are normalized into repository–pull-request records.
S2 Qualification and feature extraction. The pipeline checks whether each candidate is a runnable JavaScript/npm project and extracts project-level signals such as lockfiles, package-manager choice, scripts, browser tooling, native-addon markers, and dependency-risk features.
S3 Experiment planning. Qualified updates are converted into experiment requests by fixing the source variant, Node.js Docker image, command policy, timeout budget, and repetition count.
S4 Docker execution. Docker-backed runners perform fresh checkouts and execute install, build, and test phases while collecting structured logs, phase outcomes, and npm metadata.
S5 Outcome analysis. The analyzer filters non-attributable failures, classifies outcome reproducibility, extracts failure signatures, and maps failed npm runs to BUMP-style categories.
S6 Export. The final evidence is exported as PR-level, experiment-level, and run-level tables used in the research-question analyses.

3.1 Methodological Choices and Alternatives

Several methodological alternatives were considered before selecting the final design. The first alternative was to reproduce each project's full CI workflow exactly. This would improve ecological realism, because the executed workflow would more closely match the evidence maintainers normally observe. However, many workflows depend on private secrets, external services, project-specific infrastructure, cached state, or historical runner images that are no longer available. The chosen Dockerized Node.js setup is therefore a compromise: it cannot perfectly reconstruct every maintainer CI environment, but it makes the execution protocol explicit, repeatable, and comparable across thousands

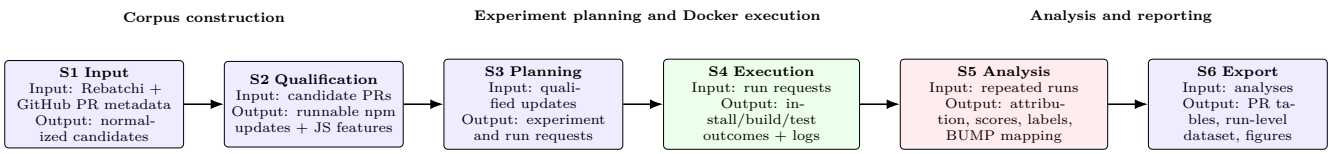

Figure 1: Methodological pipeline for constructing the npm update corpus, executing repeated Dockerized runs, classifying outcome reproducibility, and exporting PR-, experiment-, and run-level analysis artifacts. The labels above the stages show the higher-level blocks used throughout the methodology.

of projects. This choice follows prior work that uses containerization to control computational environments in reproducible software-engineering research [11, 12].

A second alternative was to run substantially more repetitions for every update. This would increase sensitivity to rare intermittent behavior and would give a stronger signal for updates whose outcomes alternate only occasionally. However, additional repetitions multiply the execution cost of an already large corpus and reduce the number of projects that can be analyzed within the available resources. The selected design therefore uses three repeated update runs as the default. This is sufficient to detect disagreement in common nondeterministic cases, while keeping the study feasible at the scale of thousands of dependency updates.

A third alternative was to execute baseline experiments for all projects rather than only when additional attribution is needed. Full baseline coverage would make it easier to distinguish update-induced failures from failures that already exist in the repository or execution environment. At the same time, it would substantially increase the number of executions, including for updates whose repeated runs already provide stable evidence. The final design therefore schedules baseline confirmation selectively, when update behavior requires additional attribution. The study also does not directly reuse the BUMP methodology as-is, because BUMP starts from known Java/Maven breaking updates [5]. Instead, BUMP is used as a comparison point for coarse failure categories, while the npm analysis keeps separate labels for reproducible success, reproducible failure, nonreproducibility, insufficient evidence, and npm-specific failure mechanisms.

3.2 Unit of Analysis and Definitions

Evidence is stored at three nested levels. A *run* is one Dockerized execution of the install, build, and test procedure. An *experiment* groups repeated runs for the same update or baseline source variant in one selected Node.js environment. An *update* is the dependency-update pull request itself. The update PR is the main unit for the paper-level empirical conclusions, while experiments and runs are used to explain environment-specific behavior, phase failures, and failure categories.

An outcome is classified as reproducible when the attributable repeated runs agree on both the relevant phase outcomes and the failure mode. A reproducible outcome can therefore be a clean pass or a stable failure. This distinction is necessary because the research question concerns stability of CI-style evidence, not whether the dependency update is behaviorally correct or acceptable to merge.

3.3 Dataset Construction

The input corpus is derived from the Rebatchi Dependabot dataset and public GitHub metadata [6]. The ingestor normalizes each row to a common schema containing an update identifier, repository name, repository URL, pull-request number, pull-request URL, default branch, provider, and ecosystem. Rows that cannot be resolved to a GitHub repository and pull-request number are discarded, and duplicate candidates are removed at the repository–PR level.

Before execution, the qualifier checks whether a candidate is a runnable npm project. It performs a Git probe, fetches the pull-request reference when it is available, and inspects package files without executing project code. A candidate is retained when the repository contains JavaScript package-management evidence, such as `package.json`, `package-lock.json`, `npm-shrinkwrap.json`, `yarn.lock`, `pnpm-lock.yaml`, or workspace configuration files. Candidates with missing package manifests, inaccessible repositories, or unavailable pull-request refs are not interpreted as dependency-update failures; they are recorded as source-access or qualification outcomes.

The normalized candidate record preserves both the pull-request reference and the repository default branch. These two source references later define the update variant and the baseline variant used by the execution pipeline.

3.4 Environment and Feature Extraction

For each qualified update, the pipeline selects one or more Node.js Docker images from repository-visible signals. The selection prioritizes declared Node requirements in `package.json`, including `engines.node` and `volta.node`; Node versions specified in GitHub Actions `setup-node` steps; and package-manager or lockfile hints. If these signals do not identify a stable requirement, the runner uses a fallback Node image. The selected Node major and the effective Node major observed in the container are stored separately so that planned and executed environments can be checked during analysis.

The qualifier extracts repository-level JavaScript features from manifests, lockfiles, scripts, and workflow files. These features include package-manager intent, lockfile

type, workspace or monorepo markers, dependency-section counts, version-range exposure, transitive dependency surface, optional and peer dependency risk, lifecycle scripts, browser-coupled test tooling, native-addon markers, frontend or end-to-end test flakiness markers, and whether the project contains a meaningful test command. Dependency-update magnitude is recorded when version pairs are available, distinguishing major, minor, and patch updates.

The runner enriches this static view with execution metadata. It records the effective package manager, selected install and test strategies, browser runtime setup, unsupported runtime markers, primary failure phase, attribution bucket, and bounded npm registry metadata for changed dependencies. The registry metadata is used to derive package popularity, package age, and maintenance activity buckets. Missing registry metadata is stored as unavailable metadata and does not change the run outcome. All these features are used for cohort analysis in RQ2, where they identify project characteristics in which non-reproducibility is concentrated. Mechanism-level explanations are then supported by the extracted phase, log, and attribution evidence.

3.5 Planning and Execution

After qualification and feature extraction, the execution protocol starts from the `updates.qualified` event. At this point the repository has already been identified as runnable, the relevant JavaScript metadata has been extracted, and the selected Node.js environment candidates are known. The planner therefore does not repeat repository analysis. Its role is to turn this qualified update description into concrete experiment scopes. Each experiment fixes the update identifier, source variant, selected Docker image, command policy, timeout budget, and repetition count. In the current configuration, an update experiment normally consists of three repeated runs for the same source variant and environment.

Each run request is treated as an independent execution attempt. The runner creates a fresh workspace, checks out the requested update or baseline source, resolves the package root, prepares the effective package manager, and executes the install, build, and test phases in order. The phase sequence is deliberately kept stable across repetitions, because the purpose is to test whether the same source state and execution policy produce the same observable outcome. During execution, the runner publishes `runs.started`, `runs.progress`, and `runs.finished` events. Separating these events makes it possible to distinguish queued runs, partially completed runs, and terminal outcomes instead of relying only on a final database row.

The command policy is applied at execution time using repository-visible configuration. Installation is resolved from the detected package manager and lockfile state. Build and test commands are selected from package scripts and, where available, GitHub Actions workflow signals. The runner can use CI-like commands, quick unit-test scripts, or browser-aware test execution when the repository indicates such requirements. Browser-related projects may receive proactive setup for Playwright, Cypress, Karma, Puppeteer, or related headless-browser runtimes. Projects without a meaningful test command are recorded as `no_tests`; they are not counted as successful test executions.

For every terminal run, the pipeline stores structured execution evidence: status, exit code, duration, Docker image metadata, install/build/test outcomes, primary failure phase, error type, effective package manager, effective Node version, retry information, log excerpts, primary failure messages, failed-test evidence, test-case inventories when extractable, and bounded npm metadata for changed dependencies. The allowed phase outcomes are `pass`, `fail`, `skip`, `no_tests`, and `unknown`. The persister writes these events into Postgres tables for updates, experiments, runs, and analyses. Flexible evidence fields are stored as JSON metadata, which allows the analysis to retain detailed run evidence without introducing a new database schema for each additional JavaScript signal.

The analyzer operates after the repeated runs for an experiment have reached a terminal state. It groups the runs by experiment identifier, recovers completed but unanalyzed experiments from Postgres after service restarts, and emits an `experiments.analyzed` event containing the reproducibility label, failure-mode summary, retry summary, JavaScript analysis fields, and BUMP-comparison fields. If the update variant shows sufficient non-reproducible evidence, the analyzer can schedule a baseline experiment for the repository's default-branch state under the same environment and command policy. This feedback path keeps baseline execution targeted at cases where a control condition is needed.

3.6 Outcome Classification

Classification uses only attributable run evidence. Runs caused by infrastructure or source-access noise, such as missing PR refs, Docker transport errors, unsupported runtimes, or local timeout-policy failures, remain in the analysis output but are excluded from the core reproducibility decision. Among the remaining runs, stable agreement in outcomes and failure signatures is classified as reproducible behavior, disagreement as non-reproducibility, and too little attributable evidence as insufficient evidence.

For each install, build, and test phase, the analyzer measures agreement across the attributable repeated runs. The reproducibility score is the fraction of runs matching the most common outcome for that phase. A score of 1.0 means that all included runs agree, while lower scores indicate disagreement across repeated executions.

The final classification uses this phase-level score together with failure-mode consistency. The classifier derives normalized failure signatures from failed phases, failed test files or specs, test-case outcome inventories, primary failure-message fingerprints, npm error codes, dependency suspects, and environment markers. An experiment is labelled `non_reproducible` when at-

tributable repeated runs disagree in a phase outcome or in one of these failure signatures. It is labelled `reproducible_clean` when the attributable runs consistently pass, and `reproducible_failure` when they consistently fail in the same way. Experiments with too few attributable runs or too little outcome evidence are assigned insufficient-evidence labels instead of being forced into a reproducible or non-reproducible class.

Retries are recorded as evidence rather than hidden from the analysis. The analyzer stores whether retries were triggered, whether they recovered a phase, and whether retries were exhausted. This makes it possible to separate clean reproducibility from outcomes that are reproducible only after recovery.

3.7 Comparison with BUMP

To answer RQ3, failed npm runs are mapped at run level to the coarse BUMP failure categories [5]. The mapping uses the primary failed phase, npm/package-manager error codes, and normalized log evidence such as failed test names, compiler messages, dependency-resolution errors, and runtime or engine constraints.

`TEST_FAILURE`. Test-phase failures, assertion failures, failed specs, snapshot mismatches, or failures from test runners such as Jest, Mocha, Karma, Cypress, Playwright, or Puppeteer.
`COMPILATION_FAILURE`. Build-phase failures, TypeScript or bundler errors, transpilation failures, native-addon compilation errors, missing build tools, or warnings-as-errors.
`DEPENDENCY_LOCK_FAILURE`. Frozen-lockfile errors, manifest/lockfile mismatches, or package-manager messages indicating that the lockfile is out of sync.
`DEPENDENCY_RESOLUTION_FAILURE`. Missing package versions, incompatible peer-dependency constraints, `ERESOLVE` errors, unavailable tarballs, registry 404 responses, or solver failures.
`ENFORCER_FAILURE`. Explicit runtime or policy rejections, such as unsupported Node.js engine constraints or package-manager version constraints.

Runs are not forced into the BUMP taxonomy when the evidence reflects source-access problems, local Docker or runner failures, unsupported setup, timeout-budget failures, clean executions, or insufficient log evidence. These cases are reported as `NOT_BUMP_COMPARABLE` or `UNKNOWN_FAILURE`. When multiple signals appear, the primary failed phase selects the candidate category and log evidence confirms or refines it.

4 Results

This section reports the analyzed corpus generated on June 20, 2026. The results are organized by research question, but the section is intended to tell one story: npm dependency-update evidence is usually stable, yet the unstable cases cluster in recognizable technical settings. Each RQ is therefore presented in three parts: why the question matters, which metric is used, and what the result shows.

Table 1: Corpus coverage used in the empirical analysis.

Stage	Count	Interpretation
Candidate PRs	3,083	Mined update PRs considered by the pipeline
Analyzed PRs	2,777	PRs with persisted analysis evidence
Repositories	2,477	Unique repositories represented by analyzed PRs
Experiments	3,142	Three-run analyses
Runs	9,426	Individual Dockerized executions
Determined experiments	2,688	Experiments with enough outcome evidence

4.1 RQ1: Reproducibility Rate

Motivation. RQ1 asks whether CI-like evidence for npm dependency updates is stable enough to support update decisions. This matters because maintainers often see only one CI result before deciding whether an update should be merged, delayed, or inspected more closely. If repeated executions of the same update state can disagree, then a single pass or fail should be interpreted as incomplete evidence rather than as a final assessment of the update.

Metric. The main units are PRs and experiments. Each experiment consists of three Dockerized executions of the same source variant, Node.js environment, and command policy; this is why 3,142 experiments produce 9,426 runs. Experiments are divided into three groups: reproducible-like, non-reproducible, and insufficient/uncertain. Reproducible-like experiments have enough attributable evidence and produce stable outcomes; this includes both repeated successes and repeated failures. Non-reproducible experiments have enough evidence, but their phase outcomes or failure signatures differ across repetitions. Insufficient/uncertain experiments are retained separately because they do not provide enough attributable evidence for a reproducibility decision.

Results. Table 1 summarizes the corpus. From 3,083 candidate PRs, the pipeline retained 2,777 analyzed update PRs across 2,477 repositories, producing 3,142 experiment analyses and 9,426 Dockerized runs. Table 2 then separates these experiments into 2,536 reproducible-like, 152 non-reproducible, and 454 insufficient/uncertain cases. Thus, the non-reproducible rate is 4.8% over all analyzed experiments and 5.7% over the 2,688 outcome-determined experiments. At PR level, 111 of 2,777 analyzed update PRs have at least one non-reproducible experiment, corresponding to 4.0%.

The stricter baseline/update definition adds a control condition. It counts an update only when the default-

Table 2: Collapsed experiment-level reproducibility outcomes.

Outcome group	Exp.	Analyzed	Determined
Reproducible-like	2,536	80.7%	94.3%
Non-reproducible	152	4.8%	5.7%
Insufficient/uncertain	454	14.4%	–

Table 3: Dominant failure phase among non-reproducible experiments.

Dominant phase	Experiments	Share
Test	71	46.7%
Install	30	19.7%
Build	26	17.1%
Unclear	25	16.4%

branch baseline is reproducible under the same environment and command policy, while the dependency-update variant is non-reproducible. This avoids treating projects that are already unstable on the baseline as update-specific evidence. Under this definition, the corpus contains 40 unique updates.

Finding for RQ1. Reproducibility is the dominant outcome, but 111 of 2,777 analyzed npm update PRs show detected non-reproducibility. A single CI result is therefore useful, but not always complete evidence.

4.2 RQ2: Phases and npm-Specific Factors

Motivation. RQ2 asks where non-reproducibility appears and which npm-specific project characteristics are associated with it. Developers, maintainers, and researchers using dependency-update evidence need to know when a CI result deserves extra caution. If instability is concentrated in particular phases or project settings, those updates may require repeated execution, manual review, or additional attribution before a CI result is treated as reliable.

Metric. The analysis uses dominant failure phase and cohort-level non-reproducibility rates. Cohorts are built from JavaScript/npm signals such as browser-coupled test tooling, native-addon markers, peer-dependency risk, package manager, Node.js major version, version-change magnitude, and package metadata. These features are interpreted as risk indicators, not as causal treatments.

Results. Among the 152 non-reproducible experiments, the test phase is the largest dominant failure phase. Table 3 shows that 71 experiments are dominated by test failures (46.7%), followed by install failures (19.7%) and build failures (17.1%). This indicates that npm non-reproducibility is not mainly an installation problem; it is most visible once projects execute tests and toolchain-heavy workflows.

The clearest project-level signal is browser coupling.

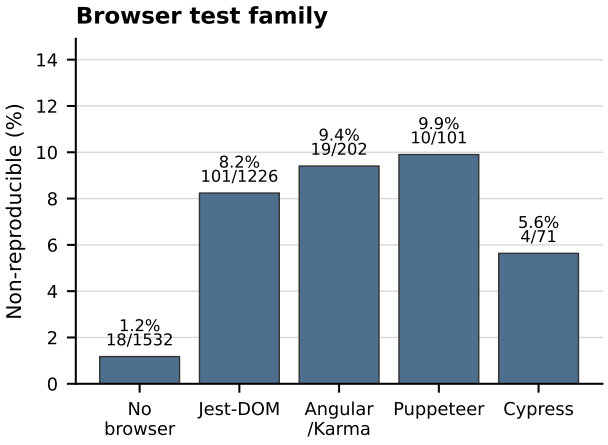

Figure 2: Experiment-level non-reproducibility rates by browser-test family.

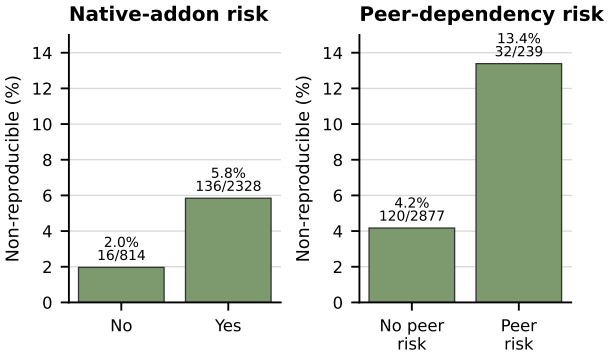

Figure 3: Experiment-level non-reproducibility rates for native-addon risk and peer-dependency risk.

Non-browser projects show 18 non-reproducible experiments out of 1,532 (1.2%), while browser-related families such as Jest-DOM, Angular/Karma, Puppeteer, and Cypress range from 5.6% to 9.9% (Figure 2). This pattern is plausible because these projects depend on headless browsers, DOM emulation, downloaded binaries, timing-sensitive tests, and test-runner services.

Native-addon and peer-dependency risk show the next strongest separation (Figure 3). Native-addon-risk projects have 136 non-reproducible experiments out of 2,328 (5.8%), compared with 16 out of 814 (2.0%) for projects without native-addon markers. The peer-dependency signal is stronger: the optional-dependency-free group with peer-dependency risk has 32 non-reproducible experiments out of 239 (13.4%), compared with 120 out of 2,877 (4.2%) without peer-dependency risk. These results suggest that non-reproducibility is associated with compiled binaries, browser environments, and dependency-graph negotiation.

Other factors provide useful context but weaker separation. Yarn experiments show a higher rate than npm experiments, 6.3% compared with 4.7%, and major updates show a higher rate than patch updates, 5.5% compared with 2.2%. Selected Node.js major ver-

sion, package popularity, package age, and maintenance activity do not show a consistent enough pattern to explain the main result. They are therefore treated as descriptive context rather than central evidence.

Finding for RQ2. Non-reproducibility is concentrated in test execution and in projects with browser coupling, native-addon risk, and peer-dependency risk. The strongest story is environmental and dependency-graph coupling, not merely update size.

4.3 RQ3: Comparison with BUMP

Motivation. RQ3 asks whether the failure categories used for Java/Maven dependency updates also explain npm failures. This matters because a benchmark taxonomy that works in one ecosystem may hide important mechanisms in another.

Metric. The comparison is based on failed-run categories, not on the same denominator as the reproducibility rates. BUMP contains 571 Java/Maven breaking dependency updates. The npm corpus contributes 5,493 failed runs for category comparison, of which 3,924 (71.4%) map to BUMP-compatible categories.

Results. Table 4 shows that npm and BUMP are very close at the coarse compilation/build level: 43.1% of npm failed runs map to compilation/build failure, compared with 42.6% in BUMP. This suggests that broad categories can support cross-ecosystem comparison. However, the similarity hides different mechanisms. The npm build/compilation group contains many native-addon compilation failures, while the test group contains browser/front-end behavior that has no direct Maven analogue. Maven-style enforcer and dependency-lock failures are almost absent in the npm mapping, while npm has a large unknown group and a meaningful set of npm-specific mechanisms.

Finding for RQ3. The BUMP taxonomy transfers only at a coarse level. npm needs additional labels for native-addon compilation, browser/front-end tests, registry or artifact availability, peer-dependency interaction, and npm-specific unknown failures.

5 Discussion

The story that emerges from the three RQs is more nuanced than simply "npm updates are reproducible" or "npm updates are flaky". In most cases, repeated execution confirmed the original evidence: the same update state kept producing the same observable outcome. This is encouraging, because it means that CI remains a useful foundation for automated dependency-update assessment. The interesting part is the boundary of this reliability. Non-reproducibility is a minority outcome, but it is concentrated in recognizable technical settings: browser-coupled tests, native addons, peer-dependency interaction, and Yarn-based installation. These are precisely the places where npm projects stop being simple package installations and start depending on external runtimes, compiled binaries, browser environments, or negotiated dependency graphs.

This distinction matters because a stable failure and an unstable failure ask maintainers to do different things. A stable failure can be debugged as an update or project compatibility problem. An unstable failure first asks whether the assessment process itself is trustworthy. In security-sensitive updates, this difference is practical rather than cosmetic: a spurious failure can delay a needed fix, while a spurious success can create false confidence in an update that deserves closer inspection.

For maintainers, the actionable conclusion is targeted confirmation rather than universal repetition. Rerunning every dependency-update PR many times would be wasteful, but updates with higher-risk characteristics should receive at least one confirmation run before being merged or rejected. Projects can also reduce instability by pinning the Node.js version, package-manager version, browser binaries, and lockfile policy where possible. The aim is not to eliminate all flakiness, but to make the evidence around dependency updates stable enough that maintainers know whether they are debugging the update or debugging the test environment.

For tool builders, the main recommendation is to expose stability as part of the update signal. Dependency-update bots and CI systems should not only report *pass* or *fail*; they should also indicate whether the result was confirmed by reruns, recovered by retries, or associated with instability-prone features. A PR that fails the same way three times is different from a PR that alternates between passing and failing. Making this distinction visible would help maintainers choose between merging, rerunning, inspecting project flakiness, debugging the dependency update, or postponing the change.

For researchers, the results suggest that future benchmarks should preserve both stable and unstable cases, but label them differently. BUMP demonstrates the value of reproducible breaking dependency updates for Java/Maven. The npm results show that JavaScript needs additional labels for browser tooling, native-addon compilation, peer-dependency interaction, registry or artifact availability, and npm-specific unknown failures. Otherwise, important mechanisms are hidden inside generic build or test categories.

These recommendations should be read together with the study's limits. The study measures detected non-reproducibility under a Dockerized protocol, not all instability that could occur in real project CI. Three repeated runs expose common instability, but can miss rare flaky behavior. External state such as package availability, browser binary downloads, native toolchains, network conditions, and host resource pressure can also affect future reruns. The observed associations should therefore be interpreted as risk indicators rather than proven causal effects.

The broader implication is that dependency-update automation should be designed around evidence that is not only fast, but also stable and interpretable. This is where the result becomes useful beyond the measured percentages: repeated execution turns a single CI outcome into a more informative signal. It separates ordinary stable success, stable failure, non-reproducibility,

Table 4: Coarse failure-category comparison between BUMP Java/Maven and the npm corpus. npm counts are failed runs mapped to BUMP-compatible categories.

Category	BUMP count	BUMP share	npm count	npm share
Compilation/build failure	243	42.6%	2,369	43.1%
Test failure	188	32.9%	1,428	26.0%
Enforcer failure	121	21.2%	9	0.2%
Dependency lock failure	14	2.5%	0	0.0%
Dependency resolution failure	5	0.9%	118	2.1%
Not BUMP-comparable	–	–	240	4.4%
Unknown failure	–	–	1,329	24.2%

and insufficient evidence. Future work should build on this distinction with adaptive repetition budgets, qualitative case studies of non-reproducible PRs, comparisons with original project CI, and replication packages that allow others to verify the reported analysis and attempt the execution protocol.

6 Responsible Research

This study uses public GitHub pull requests and public npm/GitHub metadata. No private repositories, private credentials, or intentionally collected personal data are used. Public project data still requires care, because failed builds or tests can be misread as judgements about maintainers or project quality. The paper therefore reports aggregate counts, rates, and failure categories. Individual repositories are used only to understand technical mechanisms, not to rank or criticize projects.

The main disclosure risk comes from run logs. The pipeline executes project-defined install, build, and test commands, which may print environment variables, service URLs, test data, or other repository-specific information. Raw execution logs and full per-run command output are therefore not included in the public replication package. The released artifacts instead include the source code, Docker configuration, documentation, tests, input pull-request identifiers, analysis scripts, figures, and a sanitized experiment-level result dataset.

To support reproducibility and research integrity, the experimental pipeline was implemented with automated validation checks and more than 450 tests covering data processing, execution planning, classification, and analysis logic. The implementation is documented so that the purpose and logic of the main pipeline components can be inspected.

A replication package is archived on Zenodo and cited as a separate research output [24]. It supports inspection of the pipeline, regeneration of tables and figures from the archived exports, and attempts to rerun the documented execution protocol. Future reruns may still differ because GitHub pull-request refs can disappear, npm registry responses can change, browser binaries may move, native toolchains can differ, and flaky tests may expose different behavior. The reproducibility claim is therefore about the reported analysis and documented protocol, not guaranteed bit-for-bit replay of all future Docker runs.

AI-based assistance was used for grammar, phrasing, LaTeX formatting, code navigation, suggested refactorings, and partial test generation. The author retains responsibility for the technical content, experimental design, implementation, results, and interpretation. AI-generated content is not used as experimental evidence.

7 Conclusion

This paper studied a practical weakness in automated dependency-update assessment: maintainers often rely on CI evidence, but that evidence is useful only if the same update state produces stable install, build, and test outcomes. The goal was therefore to measure whether CI-like evidence for Dependabot npm pull requests is reproducible under repeated Dockerized execution.

For RQ1, reproducibility was the dominant outcome, but not a guarantee: 111 of 2,777 analyzed PRs (4.0%) were non-reproducible. For RQ2, non-reproducibility was concentrated in test execution and was associated most strongly with browser-coupled projects, peer-dependency risk, and native-addon risk. For RQ3, the comparison with BUMP showed that broad build/compilation and test-failure categories transfer across ecosystems, but npm requires additional labels for mechanisms such as native-addon compilation, browser/front-end test behavior, dependency-graph negotiation, registry or artifact availability, and npm-specific unknown failures.

The main implication is that a single CI result remains useful, but should not always be treated as final evidence for npm dependency-update decisions. Repeated execution separates stable success, stable failure, non-reproducibility, and insufficient evidence, which are different signals for maintainers, tool builders, and researchers. This suggests that future dependency-update tools and benchmarks should record not only whether an update passed or failed, but also whether that evidence was stable under repeated execution.

References

[1] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Proc. 37th IEEE/ACM International*

Conference on Software Engineering, vol. 2, pp. 109–118, IEEE, 2015.

[2] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[3] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?", in *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 84–94, IEEE, 2017.

[4] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on GitHub Dependabot," *IEEE Transactions on Software Engineering*, vol. 49, no. 8, pp. 4004–4022, 2023.

[5] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, and M. Monperrus, "BUMP: A benchmark of reproducible breaking dependency updates," in *Proc. SANER 2024*, pp. 159–170, IEEE, 2024.

[6] H. Rebatchi, T. F. Bissyandé, and N. Moha, "Dependabot and security pull requests: Large empirical study," *Empirical Software Engineering*, vol. 29, no. 5, p. 128, 2024.

[7] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proc. 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 426–437, ACM, 2016.

[8] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software," in *Proc. 14th IEEE/ACM International Conference on Mining Software Repositories*, pp. 345–355, IEEE, 2017.

[9] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? A case study of Java projects," *Journal of Systems and Software*, vol. 183, article 111097, 2022.

[10] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2022.

[11] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[12] J. Cito and H. C. Gall, "Using Docker containers to improve reproducibility in software engineering research," in *Proc. 38th International Conference on Software Engineering Companion*, pp. 906–907, ACM, 2016.

[13] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 643–653, ACM, 2014.

[14] N. Hashemi, A. Tahir, and S. Rasheed, "An empirical study of flaky tests in JavaScript," in *Proc. IEEE International Conference on Software Maintenance and Evolution*, pp. 24–34, IEEE, 2022.

[15] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proc. 40th International Conference on Software Engineering*, pp. 433–444, ACM, 2018.

[16] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, article 202, 2020.

[17] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proc. USENIX Security*, pp. 995–1010, 2019.

[18] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Proc. 13th International Conference on Mining Software Repositories*, pp. 351–361, ACM, 2016.

[19] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proc. 14th IEEE/ACM International Conference on Mining Software Repositories*, pp. 102–112, IEEE, 2017.

[20] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[21] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? An empirical case study on npm," in *Proc. 11th Joint Meeting on Foundations of Software Engineering*, pp. 385–395, ACM, 2017.

[22] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "A large scale analysis of semantic versioning in NPM," in *Proc. 20th IEEE/ACM International Conference on Mining Software Repositories*, pp. 485–497, IEEE, 2023.

[23] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. Wiese, "I depended on you and you broke

me: An empirical study of manifesting breaking changes in client packages," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, article 94, 2023.

[24] V. Maksymiuk, "Dependency Update Reproducibility for JavaScript/npm: A Docker-Based Research Pipeline," Zenodo, 2026. doi: 10.5281/zenodo.20774694.