Mining Reproducible Cargo Dependency Updates

Reproducibility & breaking rates in the Rust ecosystem

Lyuben Todorov
Supervisors: Sebastian Proksch, Cathrine Paulsen
Examiner: Johan Pouwelse
CSE3000 Research Project
EEMCS, TU Delft

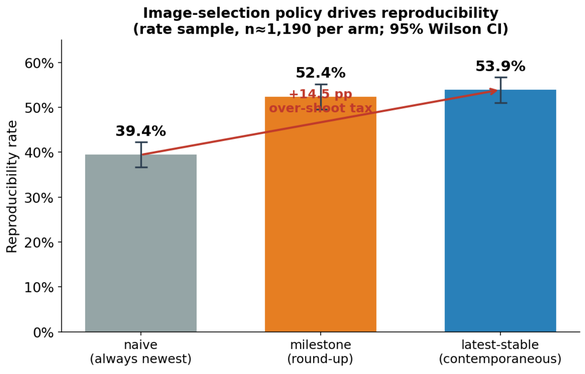
1 ▪ The problem

- Bots (Dependabot) open *millions* of dependency-update PRs.
- Dependency updates carry the risk of **breaking** their dependant?
- Java/Maven has BUMP - a curated dataset of consumer-side breaking updates, enabling researchers to study their causes, repair and prevention.
- Rust/Cargo: *no* similar dataset exists.
- Historical build reproduction is required to both reliably detect and scientifically defend such datasets.
- .. which by itself is a difficult task, as build reproductions require special environment accommodations, especially when research replication is required.

First study of breaking updates for Cargo using build reproduction for detection.

2 ▪ Environment selection policy

- Three axes to match: **OS** + package manager, **system/build deps**, **compiler**.
- A policy guesses the configuration *contemporary* with the commit.
- OS: pin the **Debian release** & `apt` **snapshot** to the commit's era; a *fat image* then adds every system dependency we detect.
- Compiler: fix OS + deps, vary it — **naive** (newest) · **milestone** (round-up) · **latest-stable** (at commit).
- Reproducibility climbs **39.4** → **52.4** → **53.9%** ($n \approx 1{,}190$); the **14.5-pp** over-shoot tax = old code on a too-new toolchain.



With latest-stable + fat-image system deps, this sample reproduces at ≈57%.

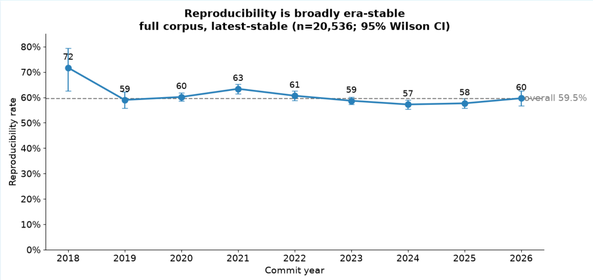
3 ▪ DURP — the pipeline

- Mine Dependabot `Bump X from A to B` PRs broadly, without an outcome filter.
- Rebuild each at its *contemporaneous* environment: rustc · Debian · apt-snapshot.
- Reproduced = environment-fingerprint match instead of byte idempotency.
- Classify every failure (6-class taxonomy).
- **20,536** PRs · live-mine 16,111 + rebatchi 4,425 (2018–2026).

Mine Dependabot PRs → Rebuild @ era env → Classify outcome

Unselective sampling ⇒ breaking updates surface naturally.

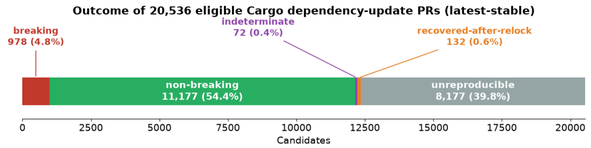
4 ▪ RQ1: How often do they reproduce?



- **12,227 / 20,536 reproduce = 59.5%** (95% CI 58.9–60.2).
- Both cohorts indistinguishable (59.5% each).
- Flat 57–63% across 2019–2026; 2018 excluded for its small sample ($n = 106$).
- Era-matching removes a fake 28-pt "old code reproduces worse" decline.

Reproducibility stable under our environment policy.

5 ▪ RQ2: What fraction break?



- **978 breaking = 8.0%** of 12,155 determinable (CI 7.6–8.5).
- Live-mine 8.8% > rebatchi 5.3% (non-overlapping CIs).
- **93% break at compile** (913); only 65 in tests.
- 21% violate semver — two-thirds of those (13% of breaks) being a stable (≥1.0) crate breaking in a minor/patch.

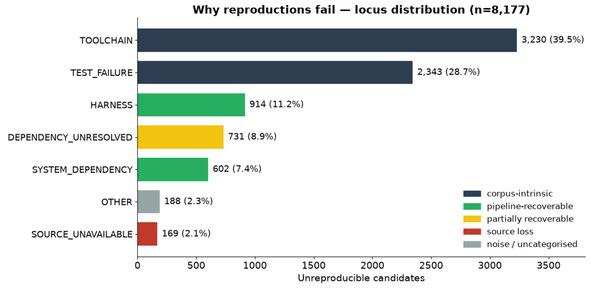
A breaking Cargo update breaks the *build*, not the test suite.

6 ▪ RQ3: Does it drift over time?

- The rate is roughly stable (RQ1), but failure *causes* rotate.
- `TOOLCHAIN` share 42.7% → 37.7%.
- Source loss (`DEPENDENCY_UNRESOLVED`) 11.2% → 6.4% — it accrues with age.
- Heavier projects over time ⇒ more OOM/timeout (`HARNESS`) failures: 9.6% → 12.4%.
- Breaking is crate-driven: outlier crates (`clap`, the `rusoto` AWS SDK) swing yearly rates 4–17%.
- Excluding outliers, breaking rates peaked at the start of our sample and have seemingly plateaued since then.

Stable rate, rotating causes: fixed recipe, moving ecosystem.

7 ▪ RQ4: Why do reproductions fail?



- 8,177 unreproducible → six classes.
- `TOOLCHAIN` 39.5% + `TEST_FAILURE` 28.6% — **corpus-intrinsic**.
- `HARNESS` 11.2% + `SYSTEM_DEPENDENCY` 7.4% — pipeline-recoverable.
- 3 blind coders: **Fleiss** $\kappa = 0.613$ (95% CI 0.49–0.72).

Most failures are intrinsic to the code, not the pipeline.

8 ▪ Insight: Rust is maturing

- Cornerstone crates reach 1.0: `tokio` (2020), `http` & `hyper` (2023).
- `NIGHTLY_REQUIRED` failures fade: 34.5% (2018) → 0.5% (2026).
- Most breaks are *signalled* by the version bump.

An eight-year record of a stabilising ecosystem.

9 ▪ Limitations

- Single architecture (`linux/amd64`).
- Dependabot-only (Renovate and human PRs excluded).
- Flakiness only lightly probed: 2.45% of single-attempt verdicts non-deterministic, on a 5% sample.
- Agreement (κ) is measured *only* at the broad 6-class level; the 38 underlying symptom-checks were authored and validated by the author alone, not expert-agreed.
- We detect breaking dependency updates at the *test* stage, meaning that breakages have managed to slip past the strict rules of the Rust compiler. The immediate follow-up question is what if it slips through the tests as well? 36% of the projects we reproduced have no test suite. This methodology is incapable of detecting the rarest, and most dangerous type of breaking update.

Honest bounds — single-arch, Dependabot-only, blind to test-passing breaks.

10 ▪ Future work

- Automating the pipeline to constantly churn and detect breakages, serving as an early-warning system for project maintainers.
- The dangerous **build-passes / tests-fail** slice — silent breaks past the type checker.
- **Cross-architecture** flakiness (beyond `linux/amd64`).
- Qualitative study of *what* actually changes in each update.
- Port DURP to **other ecosystems** — defend language-agnosticity.
- Independently **validate the classifier** (the 38 detectors).
- Automated **repair / prevention** + pre-merge risk flagging for maintainers.

Detect now; repair, predict and prevent next.