CratesBump: A Version-Bump Dataset for the Rust Ecosystem

Muhammad Hassnain
University of California, Davis
Davis, CA, USA
mhassnain@ucdavis.edu

***Abstract*—We release CratesBump, the first public dataset for predicting version bumps in the Rust ecosystem, together with SEMVER-EXTRACT, the tool that builds it. CratesBump is built from every crate on crates.io with at least two releases before 14 September 2026 (about 220,000 crates), giving over two million consecutive release pairs. Each pair has 41 features (code changes from AST diffs, dependency changes, complexity, release cadence, GitHub commit and issue activity), its source repository where listed, and two labels: the literal reading of `MAJOR.MINOR.PATCH`, and Cargo's rule, under which a `0.y` bump is a breaking change. SEMVER-EXTRACT regenerates the dataset from public crates.io data and the GitHub API. To illustrate its use we (i) benchmark bump predictors on held-out repositories, finding the task learnable but far from solved, with lower scores on later releases; (ii) measure which feature groups carry the signal; (iii) check, against declared dependency ranges, how often Cargo's rule shields dependents from pre-1.0 breaking releases; and (iv) audit a sample of releases with `cargo-semver-checks`, finding minor and patch releases that break the public API despite their version number. Dataset and tool are released under open licenses.**

***Index Terms*—semantic versioning, Rust, crates.io, dataset, mining software repositories, breaking changes, release engineering**

I. Introduction

Semantic Versioning (SemVer) [1] is a versioning scheme in which every release is numbered `MAJOR.MINOR.PATCH`: a major release makes breaking changes, a minor release adds backward-compatible features, and a patch release fixes bugs. Package managers rely on these numbers to decide whether an upgrade is safe. The specification makes no promise for `0.y.z` versions. Cargo, Rust's package manager, fills that gap [2]: the *leftmost non-zero number* marks the breaking boundary, so `0.4.5` → `0.5.0` is a major bump and `0.4.5` → `0.4.6` a minor one (npm's caret ranges do the same [3]).

Predicting the bump from the changes between two releases has been studied as a supervised learning problem on npm [4] and Ansible Galaxy [5], but to our knowledge no public bump-prediction dataset exists for Rust. Rust's SemVer checkers [6], [7] answer a related question: whether the public-API changes they detect are breaking, and so whether a release's bump is large enough. Both releases must compile, and changes in behaviour are invisible to them.

This paper contributes:

- **CratesBump** (Section III): 2,079,790 release pairs from 219,955 crates with at least two releases, each with 41 features, its source repository, and both labels; and SEMVER-EXTRACT (Section IV), the open-source tool that rebuilds it.
- **Reference usage** (Section V): we benchmark bump predictors on held-out repositories, where tree ensembles reach a macro-F1 of 0.45–0.46 (baseline 0.24) and drop to 0.40–0.42 on later releases. We measure which feature families carry the signal and show how pre-1.0 breaking releases reach dependents. Finally, we audit releases with `cargo-semver-checks`, checking every flag by hand: at least 14.9% of checked minor and 8.7% of checked patch releases break the public API.

II. Related Work

Datasets of versions and breaking changes. Several public datasets record how packages change from one version to the next, and most of them cover Java. Raemaekers et al. [8] collect Maven library versions with code metrics. Benelallam et al. [9] model Maven Central's dependencies as a temporal graph. Jezek and Dietrich [10] build a synthetic benchmark of API changes, and Reyes et al. [11] collect dependency updates that break the build. Beyond Java, Pinckney et al. [12] track every npm package version, and Rahman et al. [13] record resolved dependencies at releases of selected npm, PyPI and crates.io packages. For Rust, Schueller et al. [14] release developer, dependency and download data. None of these datasets pairs consecutive releases with change features and bump labels.

SemVer practice. Raemaekers et al. [15] found that about a third of Maven releases introduce a breaking change, while Ochoa et al. [16] found 83.4% of upgrades compliant; in Go, 28.6% of non-major upgrades break the API [17]. Bogart et al. [18] compare breaking-change policies across 18 ecosystems and Dietrich et al. [19] dependency-versioning practices across 17, both including Cargo, and Decan and Mens [20]–[22] find that package managers are more permissive about `0.y.z` than the specification.

Bump prediction. Abdalkareem et al. [4] predict the release type of npm releases from 41 features (change types, complexity, development activity) over about 6,260 releases of 31 packages. Opdebeeck et al. [5] frame the same task for 81k Ansible role version increments, and Lam et al. [23] call for "semantic version calculators" that tell whether an upgrade is safe. CratesBump brings this task to a whole ecosystem.

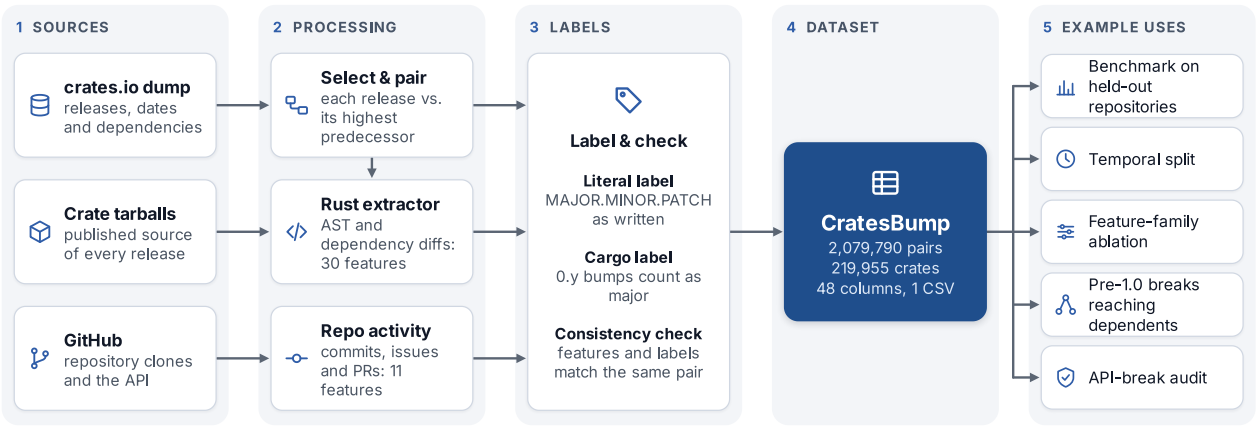


Fig. 1. How CratesBump is built (1–4) and used in Section V (5).

SemVer checkers. `cargo-semver-checks` [6] and the now deprecated `rust-semverver` [7] compare the public APIs of two builds, as APIDiff [24] does for Java. They detect breaking API changes, but only the kinds their rules cover; in Java, Sembid [25] finds 2–4 times as many APIs with behaviour breaks as with signature breaks. Nowak and Gruevski [26] ran `cargo-semver-checks` on popular crates.

Rust ecosystem. Kikas et al. [27] and Decan et al. [28] compare Cargo's dependency network with those of other ecosystems. Studies of crates.io show that unstable compiler features can break dependent crates [29] and catalogue why releases are yanked, accidental breaking changes among them [30]. Even code written entirely in safe Rust can harm its users [31], and Cargo Sherlock [32] scores supply-chain trust from crates.io metadata and analysis-tool results. None of these works labels version bumps; to our knowledge, CratesBump is the first public bump-prediction dataset for Rust.

III. The CratesBump Dataset

A. Source and scope

We use the official crates.io database dump of 12 October 2026 (Fig. 1), which lists every crate and release ever published, for crate selection, release lists, timestamps and the dependency graph. The dataset is a snapshot at 14 September 2026: it contains every crate with at least two non-yanked releases published before that instant: 225,799 of the dump's 349,979 crates (64.5%), which account for **99.4% of all crates.io downloads**. We also keep crates that never reached 1.0, since Cargo's rule gives their `0.y.z` bumps a clear meaning. They make up 84% of these crates and carry 62% of all downloads. 219,955 crates contribute at least one release pair; the others form no pair (e.g. only backports), have no tarballs left (139), or lost all their pairs to the family cap.

B. Release pairs and features

For each crate, we go through its non-yanked releases in publish order. Each release numbered above all earlier ones is paired with the highest earlier release; a *backport* (numbered below an existing release) is skipped. Publishing `1.0.0`, `1.1.0`, `1.0.5`, `1.2.0` thus yields `1.0.0`→`1.1.0` and `1.1.0`→`1.2.0`, so each diff covers exactly the changes its version number describes. A final release after its own pre-release (`1.0.0` after `1.0.0-rc.1`) forms no pair, since it has no bump class, but becomes the new highest version, so `1.0.1` is compared with `1.0.0`. 2.6% of pairs have a pre-release endpoint. So that crates with thousands of releases and generated crate families (`aws-sdk-*`, `solana-*`) do not dominate, we keep at most 100 pairs per crate and 5,000 per name family (the name up to its first - or _), sampled with a fixed seed.

For every pair we download both `.crate` tarballs and compute 41 features (Table I), following Abdalkareem et al.'s design for npm [4]: change-type counts from `syn` syntax-tree diffs of every `.rs` file, `Cargo.toml` dependency changes, complexity and size deltas, days since the previous release, and GitHub activity between the two releases. Commit features come from a clone of the crate's repository, issue and pull-request features from the GitHub API, so they reflect the repository as it was when crawled (October–November 2026). 84.0% of pairs have at least one non-zero GitHub feature. The others are all zero because the crate lists no repository (5.2%) or one outside GitHub (3.9%), its GitHub repository was deleted or made private (4.0%), or nothing in the repository falls in the window (2.9%: half quick re-releases, half histories that miss the release).[1]

C. Bump labels: literal reading vs. Cargo's rule

The literal label `release_type_naive` compares the three numbers as written; the Cargo label `release_type` applies Cargo's rule, under which the leftmost non-zero component marks a breaking change. Both labels compare `version` with `prev_version`. We apply a consistency check against the crates.io dump, which confirms that, for

[1]Two known bugs, to be fixed in the next release, add to the 4.0%: unusual GitHub URLs (14 pairs) and unchecked `git log` failures (≤49 repos).

TABLE I

THE 48 COLUMNS OF CRATESBUMP, ONE ROW PER RELEASE PAIR (PREV_VERSION → VERSION). THE 41 FEATURE COLUMNS (LOWER PART) DESCRIBE WHAT CHANGED BETWEEN THE TWO RELEASES.

Group	Cols	Type	What the columns hold
Pair key	3	text	`package`, `prev_version`, `version`
Repository	1	text	`repository`, e.g. `github.com/serde-rs/serde`
Labels	2	label	major, minor or patch, by Cargo's rule and by the literal reading
Publish time	1	time	when `version` was published (UTC)
Code (AST)	20	count	files, functions, blocks added/deleted
Dependencies	4	count	dependencies added/removed/updated
Complexity, size	5	number	complexity change, lines added/deleted
Release timing	1	number	days since `prev_version`
GitHub commits*	7	number	commits, authors, bug-fix commits
GitHub issues*	4	count	issues and PRs opened/closed

*Counted between the two publish times; 0 when no GitHub data exists.

TABLE II

PAIRS ON WHICH THE LITERAL AND CARGO LABELS DISAGREE, BY KIND OF VERSION CHANGE: ALL PRE-1.0, 1,712,585 PAIRS (82.3%).

Version change: literal → Cargo label	Pairs	Share
`0.y.z`→`0.y.(z+1)`: patch → minor	1,033,349	60.3 %
`0.y.z`→`0.(y+1).0`: minor → major	555,772	32.5 %
`0.0.z`→`0.0.(z+1)`: patch → major	109,239	6.4 %
`0.0.z`→`0.y.z`: minor → major	14,225	0.8 %

every pair, the features and labels describe the same two releases. The labels coincide when MAJOR ≥ 1 and differ on every pair whose new version is below 1.0.0 (82.3% of pairs). Table II breaks the differences down. Cargo labels are 35.4 / 55.2 / 9.3% major / minor / patch, against 2.8 / 32.9 / 64.3% for literal labels.

Storage and schema. CratesBump is one gzip-compressed UTF-8 CSV file (`data/cratesbump.csv.gz`) with one row per release pair. Rows are identified by `package` and `version`, written as in the crates.io dump, so users can link them to crates.io's own data such as downloads or dependencies. Only `repository` can be empty. The crate-to-repository map, the audit results, the build reports and `SCHEMA.md`, which describes every column, ship alongside.

IV. THE SEMVER-EXTRACT EXTRACTION TOOL

SEMVER-EXTRACT is the open-source pipeline in Fig. 1; it lives in the `CratesBump` repository [33]. Its Rust part diffs the `syn` syntax trees and `Cargo.toml` dependencies of both tarballs of a pair; Python scripts pair the releases, attach labels and repositories, and add the GitHub features. A `Makefile` runs everything (Section VII). It needs stable Rust, Python 3.11+, the crates.io dump and a GitHub token.

SEMVER-EXTRACT is built for whole-ecosystem runs on one machine. It reads release lists from the dump rather than the rate-limited API, so it only downloads tarballs; 16 parallel workers write one CSV per crate, and a restart skips finished crates. The GitHub step queries 25 repositories per GraphQL call and caches every response. A full rebuild takes about two days, mostly GitHub crawling under its hourly quota; a later snapshot needs only a newer dump and cutoff date.

TABLE III

PREDICTING THE CARGO BUMP LABEL (MAJOR, MINOR OR PATCH) OF PAIRS FROM REPOSITORIES NOT SEEN IN TRAINING: MEAN ± S.D. OVER FIVE REPOSITORY SPLITS. MACRO-F1 AVERAGES THE THREE LABELS' F1; MCC IS 0 FOR RANDOM GUESSING AND 1 FOR PERFECT PREDICTION. THE FIRST TWO ROWS IGNORE THE FEATURES; THE BEST SCORE IS IN BOLD.

Model	Macro-F1	MCC
Always "minor" (baseline)	0.236 ± 0.007	0.000 ± 0.000
Random by label share (baseline)	0.333 ± 0.001	0.000 ± 0.001
Random Forest	0.448 ± 0.007	0.216 ± 0.012
LightGBM	0.447 ± 0.007	0.223 ± 0.012
XGBoost	0.457 ± 0.004	0.239 ± 0.011
Hist Gradient Boosting	**0.459** ± 0.005	**0.241** ± 0.011
Gradient Boosting	0.449 ± 0.005	0.225 ± 0.014
Decision Tree	0.437 ± 0.008	0.192 ± 0.011
Logistic Regression	0.361 ± 0.036	0.144 ± 0.023

V. REFERENCE USAGE

A. Benchmark on held-out repositories

We train seven classifiers to predict the Cargo label from the 41 features and compare them with two baselines that ignore the features: one always predicts "minor", the most common label; the other picks each label at random, as often as it occurs in the data.

To test on unseen projects, we hold out 20% of the source *repositories*, so crates from one repository are all in training or all in test (crates without one form their own group). We repeat this with five random splits and balance only the training data (SMOTE).

We tuned each model by random search (30 settings, 3-fold cross-validation by repository) on about 500,000 pairs. This sample overlaps the test repositories, but the best tuned model scores no higher than the best untuned one (both 0.459). We report macro-F1 and the Matthews correlation coefficient (MCC; 0 for random guessing, even with imbalanced classes).

The five tree ensembles reach macro-F1 0.447–0.459 (MCC 0.22–0.24), well above both baselines (Table III); patch releases, 9% of pairs, are the hardest class (F1 about 0.17). We also find that holding out single crates instead of whole repositories inflates scores, because crates of one repository share its GitHub features and release dates. The more a model can memorise, the larger the effect: the tuned Random Forest rises from 0.448 to 0.461, but with the original, deeper settings from 0.455 to 0.612, so splits should group by `repository`.

B. Temporal split

Trained on repositories whose pairs all predate 22 June 2026 (the 80th percentile of release times) and tested on those whose pairs all follow it (927,421 and 167,154 pairs), the tree models reach macro-F1 0.399–0.423, below their 0.437–0.459 on held-out repositories. The test repositories are young, so the drop mixes a shift over time with one towards less mature crates.

C. Which feature families carry the signal?

On the first held-out split (Random Forest, macro-F1 0.451), we shuffle each feature family in the test set five times and record the mean macro-F1 drop: code change types $+0.029$, dependencies $+0.014$, complexity and size $+0.012$, days since the previous release $+0.011$, GitHub commits $+0.008$, and GitHub issues and pull requests $+0.004$. As on npm, where change types were the strongest predictors [4], code features carry the most signal and GitHub features the least, although the original unbounded-depth Random Forest leaned on GitHub more ($+0.046$, $+0.027$).

D. What lets pre-1.0 breaking releases reach dependents?

A release such as `0.4.5` → `0.5.0` looks minor but is breaking under Cargo's rule. Do the crates that depend on it pick it up automatically? Usually not: by default, a dependency on `0.4.5` accepts updates only below `0.5.0`, and this default covers 94.9% of the 33.6 M regular dependency declarations in the crates.io dump. For each of the 569,997 such releases in CratesBump, we check whether the requirement of every crate that depended on the package just before the release accepts it (ignoring pre-release tags). Only 26,558 of 1,200,628 dependency links (2.2%) would have picked up the breaking release, and only 8,525 of these releases (1.5%) reached any dependent. Almost all such links use loose requirements: any `0.y.z` version (e.g. `0.*`; 58%), no upper bound (e.g. `>=0.22`; 24%) or any version (`*`; 15%).

E. External API audit

Do minor and patch releases break their public API? We checked a random sample with `cargo-semver-checks` [6] 0.51.0, which compares the public APIs of two versions using rules called lints and says which kind of release the changes require. We drew 450 pairs, 150 per Cargo label and each from a different crate, and built them offline in a container.

The tool could check 305 of them. The rest are command-line programs or macro crates it cannot read (71), or no longer build or resolve (74; eight only because the tool turns on conflicting features). Older releases fail more often.

The tool flagged 21 of 101 minor and 10 of 103 patch releases as needing a major release. We read the code of all 31: 24 are real breaks, 6 are unlikely to matter (e.g. generated bindings), and 1 is not a break. So **at least 14.9% of minor and 8.7% of patch releases break their public API** (95% CIs 9.2–23.1% and 4.7–15.8%); in 20 of the 24, ordinary code using the crate stops compiling.

Some caveats. The rates are lower bounds, since the tool misses behaviour changes and most type changes. Most checked minor releases (92 of 101) are `0.y.z`, so the minor rate mostly describes pre-1.0 crates. One break is a pre-release; without pre-releases the rates are 14.3% and 9.0%. With one release per crate, the rates describe a typical crate, not all releases.

Our rates are well above the 3.2% that Nowak and Gruevski [26] found with an early version of the tool on the 1,000 most-downloaded crates (ours combined: 11.8%). The newer lints explain little of the gap: with only the 45 lints of mid-2023, our rates drop to 12.9% and 6.8%. Their study also used default features only, confirmed each break with failing code, and checked every release of popular crates; we have not tested which difference matters.

VI. Limitations and Future Work

Labels and GitHub features. Labels are declared bumps: Cargo's rule fixes how a version change is read, not whether the maintainer chose correctly (Section V-E). GitHub features reflect each repository as crawled and are shared by its crates, hence the held-out-repository splits. A zero can mean "no data" or "no activity" (Section III-B).

Caps, snapshot and analyses. Capped crates contribute a random sample of their pairs (SEMVER-EXTRACT can re-extract them in full), and releases from 14 September 2026 on are missing. The audit covers only library crates that still build, and the dependents analysis builds nothing.

Future research. CratesBump can help study which changes maintainers under-version and whether a model can flag them before publishing, when and why crates leave `0.y.z`, and whether bump predictors transfer across ecosystems and time.

Future improvements. Later releases could add a `cargo-semver-checks` label per buildable pair, regular snapshots, changelog and test signals, non-GitHub hosts and a missing-data flag.

VII. Availability

CratesBump and SEMVER-EXTRACT are on GitHub [33] as release `v1.0.0`, with the code under the MIT license and the dataset under CC BY 4.0; before the camera-ready version, we will archive this release on Zenodo and add its DOI. The repository's `Makefile` regenerates every table and number in this paper except three GitHub-coverage details that need the crawl caches (see the README): the benchmark and temporal split need only the CSV file; the other analyses also need the crates.io dump, a GitHub token and, for the audit, Docker (see the README). Rebuilding the dataset (`make rebuild-full`) takes about two days; results with the original model settings are in `results/original_params/`. Following the call's FAIR principles, CratesBump is findable (a tagged release; a Zenodo DOI at camera-ready), accessible (open licenses), interoperable (one CSV file) and reusable (data dictionary, schema, `Makefile`). *Prior use:* this paper is the first description of CratesBump and SEMVER-EXTRACT; neither has been used in earlier publications.

Acknowledgment

Per IEEE policy, we disclose that Claude (Anthropic), via Claude Code [34], helped write the code and draft and polish the text; the author verified all of it and takes full responsibility.

References

[1] T. Preston-Werner, "Semantic versioning 2.0.0," https://semver.org/, accessed: 2026-10-12.

[2] The Cargo Team, "The Cargo book: Specifying dependencies," https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html, accessed: 2026-11-05.

[3] npm, Inc., "node-semver: The semantic versioner for npm," https://github.com/npm/node-semver, caret ranges. Accessed: 2026-11-04.

[4] R. Abdalkareem, M. A. R. Chowdhury, and E. Shihab, "A machine learning approach to determine the semantic versioning type of npm packages releases," arXiv preprint arXiv:2204.05929, 2022.

[5] R. Opdebeeck, A. Zerouali, C. Velázquez-Rodríguez, and C. De Roover, "On the practice of semantic versioning for Ansible Galaxy roles: An empirical study and a change classification model," *Journal of Systems and Software*, vol. 182, p. 111059, 2021.

[6] P. Gruevski, "cargo-semver-checks: Scan your Rust crate for semver violations," https://github.com/obi1kenobi/cargo-semver-checks, version 0.51.0. Accessed: 2026-11-04.

[7] The Rust Project, "rust-semverver: Automatic checking for semantic versioning in library crates," https://github.com/rust-lang/rust-semverver, archived repository. Accessed: 2026-11-04.

[8] S. Raemaekers, A. van Deursen, and J. Visser, "The Maven repository dataset of metrics, changes, and dependencies," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 221–224.

[9] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The Maven dependency graph: A temporal graph-based representation of Maven Central," in *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2019, pp. 344–348.

[10] K. Jezek and J. Dietrich, "API evolution and compatibility: A data corpus and tool evaluation," *Journal of Object Technology*, vol. 16, no. 4, pp. 2:1–23, 2017.

[11] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, and M. Monperrus, "BUMP: A benchmark of reproducible breaking dependency updates," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2024, pp. 159–170.

[12] D. Pinckney, F. Cassano, A. Guha, and J. Bell, "npm-follower: A complete dataset tracking the NPM ecosystem," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 2132–2136.

[13] I. Rahman, J. Marley, R. Paramitha, and L. Williams, "CHRONO-RESOLUTION: A dependency resolution dataset at release points for npm, PyPI, and crates.io packages," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2026, pp. 1184–1188.

[14] W. Schueller, J. Wachs, V. D. P. Servedio, S. Thurner, and V. Loreto, "Evolving collaboration, dependencies, and use in the Rust open source software ecosystem," *Scientific Data*, vol. 9, no. 1, p. 703, 2022.

[15] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[16] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? Semantic versioning and impact of breaking changes in Maven Central," *Empirical Software Engineering*, vol. 27, no. 3, p. 61, 2022.

[17] W. Li, F. Wu, C. Fu, and F. Zhou, "A large-scale empirical study on semantic versioning in Golang ecosystem," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1604–1614.

[18] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 4, pp. 1–56, 2021.

[19] J. Dietrich, D. J. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2019, pp. 349–359.

[20] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, 2021.

[21] ——, "How magic is zero? An empirical analysis of initial development releases in three software package distributions," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW)*, 2020, pp. 695–702.

[22] ——, "Lost in zero space – An empirical comparison of 0.y.z releases in software package distributions," *Science of Computer Programming*, vol. 208, p. 102656, 2021.

[23] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the semantics into semantic versioning," in *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2020, pp. 157–179.

[24] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "APIDiff: Detecting API breaking changes," in *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 507–511.

[25] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? Static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[26] T. Nowak and P. Gruevski, "Semver violations are common, better tooling is the answer," https://predr.ag/blog/semver-violations-are-common-better-tooling-is-the-answer/, Sep. 2023, accessed: 2026-11-05.

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

[28] 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.

[29] C. Li, Y. Wu, W. Shen, Z. Zhao, R. Chang, C. Liu, Y. Liu, and K. Ren, "Demystifying compiler unstable feature usage and impacts in the Rust ecosystem," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024.

[30] H. Li, F. R. Cogo, and C.-P. Bezemer, "An empirical study of yanked releases in the Rust package registry," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 437–449, 2023.

[31] M. Hassnain and C. Stanford, "Counterexamples in safe Rust," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2024, pp. 128–135.

[32] M. Hassnain, A. Basu, E. Ng, and C. Stanford, "Cargo Sherlock: An SMT-based checker for software trust costs," in *Proceedings of the IEEE/ACM 14th International Conference on Formal Methods in Software Engineering (FormaliSE)*, 2026.

[33] M. Hassnain, "CratesBump: a version-bump dataset for the Rust ecosystem, and the semver-extract tool that builds it," https://github.com/muhammad-hassnain/CratesBump, 2026, release v1.0.0.

[34] Anthropic, "Claude Code," https://claude.com/product/claude-code, 2026, accessed: 2026-11-10.