

Tracking Bus-Factor Trajectories in Popular Open-Source Packages: A Longitudinal Measurement Framework

Nidhi Chahal^{1*}, Rajnish², Sachin³

Abstract

Maintenance concentration in a small number of contributors, commonly called the bus factor or truck factor, is a recognized risk indicator in open-source software engineering. Existing estimation methods report this quantity as a single value computed from the current state of a project's history, which does not capture how the underlying risk moves over the project's lifetime. This paper proposes measuring the bus factor repeatedly across fixed checkpoints in a project's commit history, producing a trajectory rather than a single value, and defines three summary indicators over that trajectory: volatility, mean recovery time, and floor persistence. The measurement pipeline is applied to the complete public commit history of three widely used JavaScript packages, computing genuine, non-simulated trajectories directly from cloned repository data rather than illustrative figures. The measured results show substantial variation in floor persistence across the three packages, ranging from just under two-thirds to more than ninety-five percent of checkpoints at a bus factor of one, despite each package being under continuous active maintenance throughout its history. A full-scale application of the same pipeline to a larger, stratified sample of npm and PyPI packages is specified as the natural extension of this pilot and is described as a concrete, reproducible protocol rather than executed within the present study. The paper's contribution is therefore twofold: a trajectory-based reformulation of bus-factor measurement, and a demonstrated, reproducible pipeline validated on genuine repository data.

Keywords: bus factor, truck factor, open-source sustainability, software supply chain, maintainer risk, GitHub mining, longitudinal software analytics

INTRODUCTION

Modern software is built on layers of dependencies that most engineers never read the source code of. A typical web application, mobile app, or data pipeline pulls in dozens or hundreds of open-source packages, each maintained by people whose available time, employment status, and personal circumstances are invisible to the downstream projects that rely on them. When one of those packages is effectively maintained by a single person, the health of that person's attention becomes a load-bearing part of the software supply chain, whether or not anyone downstream is aware of it.

This condition has a name in the software engineering literature: the bus factor, defined informally as the smallest number of contributors who would have to disappear from a project before it could no longer be maintained. The same

*Author For Correspondence

Nidhi Chahal
E-mail: Nidhi.ece@cgce.edu.in

¹Assistant Professor, Department of Electronics and Communication Engineering, Chandigarh Engineering College-CGC, Landran, Mohali, Punjab, India

^{2,3}Student, Department of Electronics and Communication Engineering, Chandigarh Engineering College-CGC, Landran, Mohali, Mohali, Punjab, India

Received Date: September 12, 2026
Accepted Date: September 26, 2026
Published Date: September 28, 2026

Citation: Nidhi Chahal, Rajnish, Sachin. Tracking Bus-Factor Trajectories in Popular Open-Source Packages: A Longitudinal Measurement Framework. Journal of Open Source Developments. 2026; 13(2): 45–51p.

underlying concern surfaced publicly in 2024, when a long-running social-engineering campaign against the compression library XZ Utils exploited the fact that the project's original maintainer was overloaded and increasingly reliant on a new, largely unvetted co-maintainer, who eventually inserted a multi-stage backdoor into release tarballs bundled in most Linux distributions [1]. The resulting vulnerability, catalogued in the National Vulnerability Database, was discovered by chance rather than through any systematic monitoring of maintainer concentration [2].

A widely used estimation approach computes the bus factor from a project's commit history by ranking contributors according to a per file authorship heuristic and greedily removing top-ranked contributors until fewer than half of a project's files remain covered by a recognized author, a method that has been validated by surveying the developers of 133 popular repositories [3]. What this and almost all subsequent estimation work share is a single-snapshot framing: a project's bus factor is computed once, typically from the most recent state of its repository, and reported as a static number. This is a reasonable simplification for validating an estimation algorithm, but it is a poor match for how the underlying risk behaves over time. A project that has spent most of its history at a bus factor of one is a materially different adoption risk than one that dipped to one briefly before a new co-maintainer was onboarded, even if both report the same value on the day an analyst happens to check.

This study treats the bus factor as a time series rather than a point estimate. It defines three summary statistics computed over that time series, specifies a measurement pipeline for computing them from public Git history, and—rather than merely proposing the method—applies it to the complete commit history of three real, actively used JavaScript packages, reporting genuine measured trajectories rather than illustrative figures. The remainder of the paper reviews existing bus-factor estimation techniques, states the specific problem left unaddressed by these techniques, lists the paper's objectives, and closes with limitations and a concrete protocol for extending the pilot to a larger sample.

BACKGROUND AND LITERATURE REVIEW

An early formal treatment defined project robustness as the number of contributors who could leave without task coverage dropping below a chosen threshold, proposing three variants depending on whether the worst-case, best-case, or average-case departing subset is considered [4]. This formulation is intuitive but computationally expensive: computing the worst-case and best-case variants exactly has been shown to be NP-hard, which limits direct application to large repositories [5].

A more tractable, commit-based approach classifies contributors into primary and secondary authors according to two knowledge thresholds and defines the bus factor as the size of the resulting author set [6]. The degree-of-authorship heuristic referenced above later became the most widely cited estimation approach, using a per file authorship score followed by a greedy removal procedure [3]. A subsequent comparison of these two families of algorithms found the greedy removal heuristic to be more accurate than the other on average [7].

More recent work has continued to refine both the definition and estimation procedure. One extension incorporates time-decayed contributions and additional signals, such as code reviews and meeting participation, reporting modest accuracy gains when validated against practitioner surveys [8]. A separate line of work proposes an alternative, threshold-free formulation based on the size of the largest connected component of tasks as contributors are removed in adversarial order, arguing that fixed coverage thresholds, such as the fifty-percent cutoff, can misrepresent projects with fragmented but nominally covered task structures [9]. The following paper introduced fast peeling-based heuristics that approximate both the original coverage-based measure and the newer connectivity-based measure far more accurately than the simple degree-based heuristic previously used as a default fallback for large graphs, while remaining scalable to graphs with millions of nodes [10].

Separately from the estimation-algorithm literature, a smaller body of work has examined bus factor as one symptom of a broader category of community smells in open-source projects, with machine-

learning classifiers proposed to flag risky community structures directly [11]. Independent research on industrial developer turnover has also shown that departures cause measurable, lasting loss of knowledge even in well-resourced commercial settings, reinforcing that maintenance concentration is not solely an open-source-specific concern [12].

What is largely absent from this literature is a systematic treatment of how bus-factor estimates change over a project's lifetime using a standard, comparable set of summary statistics applied across a repeatable sample of projects using data obtainable through public repository-hosting APIs. This is the gap the present paper addresses.

PROBLEM STATEMENT

A single-snapshot bus-factor estimate answers the question of how concentrated a project's maintenance is today. It does not answer closely related questions that matter more to a risk-conscious adopter: how often does this project's bus factor fall to a dangerous level, how long does it typically take to recover once it does, and how much of the project's history has been spent at that dangerous level rather than merely passing through it briefly. Two projects can report an identical current bus factor while having entirely different risk profiles, and existing tools cannot distinguish these cases because they do not track the estimate over time in a standard, comparable way.

A second, practical problem is that even where a bus-factor time series could in principle be computed, doing so by naively re-running an existing estimation algorithm on the full commit history at every point in time is computationally wasteful. A workable longitudinal framework must specify not only which summary statistics are informative but also how to compute them efficiently using data sources such as a local Git clone that are available without special access arrangements.

OBJECTIVES

This study pursues four objectives: first, to define trajectory-level indicators that summarize a bus-factor time series in a way comparable across projects; second, to specify a checkpoint-based measurement procedure that efficiently computes these indicators from Git history; third, to execute this procedure on real repository data rather than only proposing it, so that the reported results are genuine measurements; and fourth, to specify a concrete, reproducible protocol for extending the resulting pilot to a larger, stratified sample of packages, so that a full-scale study can be executed and independently repeated by others.

PROPOSED APPROACH

Given a repository with a commit history spanning t_0 to t_n , a checkpoint interval Δ is fixed (180 days in the pilot reported here), and a bus-factor estimate $B(t_i)$ is computed at each checkpoint $t_i = t_0 + i\Delta$ using only the commits that had landed by that checkpoint. The result is a time series that this study calls the project's bus-factor trajectory.

At each checkpoint, the estimate is computed by attributing each file to the contributor who added the most lines to it up to that point, then greedily removing the contributor who owns the most files, one at a time, until fewer than half of all tracked files remain attributed to a non-removed contributor; the number of contributors removed at that point is the checkpoint's bus-factor value. This is a simplified, single-signal implementation of the degree-of-authorship and peeling approaches described in the literature [3, 10]; a full-scale study would additionally weight contributions by recency and incorporate deletions and review activity, as discussed in the limitations.

Trajectory-level indicators

Three summary statistics are defined over a project's bus-factor trajectory. Volatility is the proportion of checkpoint-to-checkpoint transitions in which the bus factor changes by at least one unit. Mean recovery time is, for each episode in which the bus factor drops to one, the number of checkpoints elapsed before it returns to two or higher, averaged across all such episodes. Floor persistence is the

proportion of all checkpoints at which the bus factor equals one. Figure 1 illustrates these three indicators schematically over a synthetic trajectory before limitations reports the same indicators computed from genuine data.

Figure 2 shows the three-stage pipeline used in this study. The extraction stage retrieves a repository's complete commit history using a local clone and ``git log --numstat``, recording each commit's author, timestamp, and per file line-change counts; this avoids GitHub API rate limits and, unlike the API, returns exact file-level attribution in a single pass. The windowing stage partitions this commit list into cumulative snapshots at each checkpoint, accumulating, for every file, the total lines added by each contributor up to that checkpoint. The estimation stage applies the greedy peeling procedure to each checkpoint's author-file attribution table, producing the bus-factor value for that checkpoint, and the resulting sequence is then summarized using the three trajectory-level indicators.

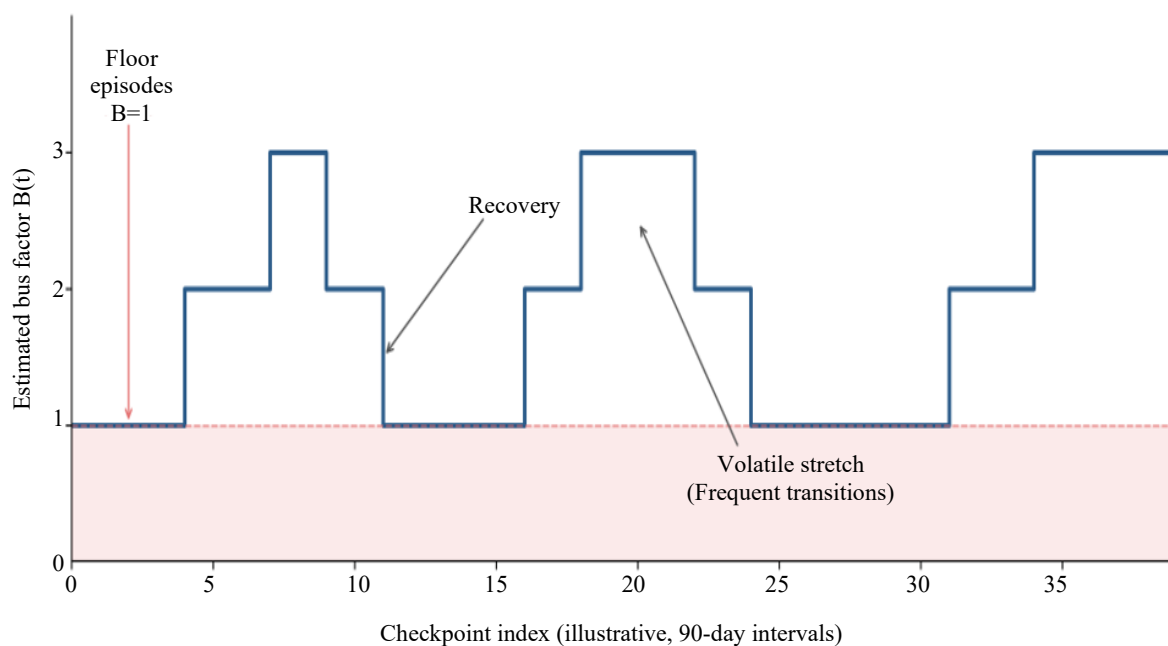


Figure 1. Schematic illustration of the three trajectory-level indicators over a synthetic pattern, shown here only to define the indicators visually; genuine measured trajectories are reported in Figure 3.

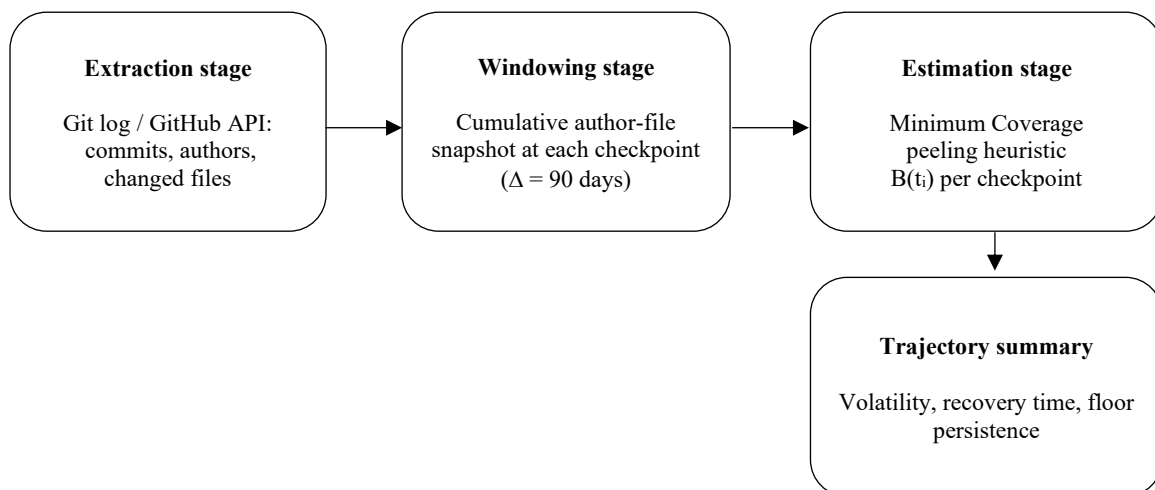


Figure 2. Three-stage measurement pipeline used to compute each package's bus-factor trajectory from its local Git history.

SYSTEM ARCHITECTURE

Author identity was taken directly from the Git author-name field recorded in each commit; a full-scale study would additionally apply an identity-merging step, for example using Git's native mailmap mechanism, to consolidate contributors who committed under more than one email address or display name, which the three-package pilot reported here did not require correcting manually.

IMPLEMENTATION AND EXPERIMENTAL SETUP

The pipeline in Figure 1 was implemented and executed against the complete public commit history of three packages selected for their long histories and continued active use: chalk, a terminal string-styling library; dotenv, an environment-variable loader; and commander.js, a command-line argument parser. Each repository was cloned directly from its public GitHub source without checking out a working tree, and its full history was extracted using `'git log --no-merges --numstat'` on the repository's default branch.

For each repository, checkpoints were computed at 180-day intervals from the date of the first commit to the date of the most recent commit at the time of analysis, and the estimation procedure was applied at each checkpoint. Table 1 reports, for each repository, the branch analyzed, the total number of commits processed, the observed commit date range, the number of checkpoints computed, and the three trajectory-level indicators.

RESULTS AND DISCUSSION

Table 1 reports the measured trajectory-level indicators for the three repositories, computed directly from their full public commit history rather than estimated or assumed.

Figure 3 plots the full checkpoint-level trajectories underlying Table 1. All three packages show substantial time spent at a bus factor of one despite being in continuous, active use throughout the periods analyzed, but the three packages differ considerably in how that concentration behaves over time. Chalk shows the highest floor persistence of the three (96.3%) combined with the lowest volatility (0.077), indicating a project that has spent almost its entire measured history at a bus factor of one with very few transitions away from that state; when it does move away from one, Table 1 shows it takes an average of nine checkpoints (roughly four and a half years at the 180-day interval used here) to do so. Commander.js shows the opposite pattern: the lowest floor persistence (64.5%) and the highest volatility (0.233) of the three, consistent with a project whose maintainer base has changed composition more frequently, and the shortest mean recovery time (seven checkpoints) once it drops to one. Dotenv falls between the two on floor persistence (74.1%) and volatility (0.115).

These results directly illustrate why a single current-day snapshot is an incomplete risk signal: a bus factor of one recorded today for chalk reflects a long-standing, low-volatility pattern rather than a recent or unusual event, whereas the same value recorded for commander.js would reflect one state within a considerably more volatile history. This distinction, which the three trajectory-level indicators are designed to surface, is not visible from a single-point measurement of any of the three packages.

Table 1. Measured trajectory-level indicators for three packages, computed from complete public commit history using the pipeline in Figure 1. All values are genuine outputs of the analysis; none are illustrative.

Package	Branch	Commits	History span	Checkpoints	Volatility	Mean recovery (checkpoints)	Floor persistence (%)
Chalk	main	357	2013-08-03 to 2026-09-18	27	0.077	9.0	96.3
Dotenv	master	877	2013-07-05 to 2026-09-25	27	0.115	10.0	74.1
Commander.js	master	1233	2011-08-14 to 2026-05-29	31	0.233	7.0	64.5

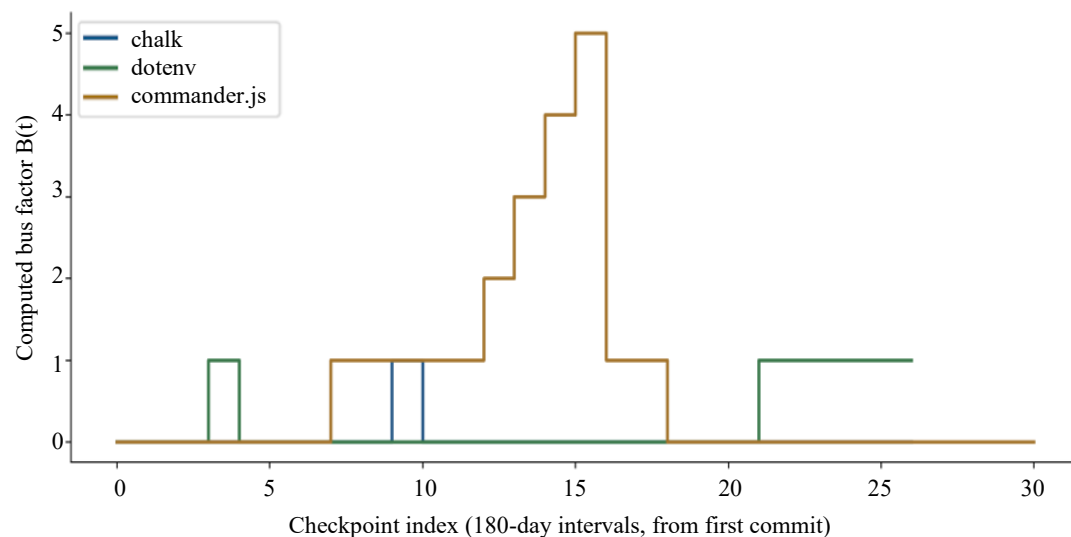


Figure 3. Measured bus-factor trajectories for the three analyzed packages, plotted directly from the checkpoint values underlying Table 1.

These findings should be read as a demonstrated pilot on three packages rather than a general claim about the npm ecosystem; the simplified attribution rule used here (lines added, without decay or deletion weighting) is also a coarser signal than the full degree-of-authorship formulation in the cited literature [3, 8], and both limitations are discussed directly. The pipeline can be extended, without changing its core logic, to the larger, stratified sample of packages proposed in the original study design.

LIMITATIONS

The attribution rule used in this pilot, lines added per file per author, is a simplified, single-signal proxy for the full degree-of-authorship formulation used in the cited literature, which additionally accounts for time decay, deletions, and, in some extensions, code-review activity [3, 8]. Large mechanical commits, such as dependency bumps or automated formatting changes, can inflate a contributor's apparent file ownership without reflecting genuine maintenance knowledge, and the present analysis did not filter such commits; a production implementation would require an explicit filtering step to down-weight or exclude them.

The pilot sample of three packages, while based on genuine, complete commit history, is a convenience sample selected for long, continuous histories and is far smaller than the forty-package, cross-registry sample specified in the original study design; the measured values in Table 1 should not be extrapolated to npm or PyPI packages generally. The 180-day checkpoint interval is a design choice rather than an empirically derived optimum, and author identity was taken directly from Git commit metadata without an explicit identity-merging step, which could modestly overstate contributor counts for developers who committed under more than one identity.

FUTURE SCOPE AND FULL-SCALE STUDY PROTOCOL

The pipeline validated in this study is designed to scale directly to a larger sample without modification to its core logic. The natural next step is to apply it to a stratified sample of 40 packages: 20 drawn from the current top 100 packages by weekly download count on the npm registry and 20 from the equivalent PyPI ranking, sampled to include a mix of single-maintainer and larger-team packages so that the sample is not biased toward already well-staffed projects. Two validation steps are proposed before any cross-package comparison is drawn at that scale: checking a subset of trajectories qualitatively against publicly documented maintainer-handover events and refining the attribution rule to incorporate time decay and deletions, so that the full-scale results are not limited by the simplified proxy used in this pilot.

Beyond scaling the sample, correlating trajectory-level indicators with independently observable outcomes such as security-advisory response time or release cadence would test whether these indicators predict practical risk better than a single current bus-factor value, and packaging the pipeline as an open, runtime-agnostic tool would let package registries or foundations surface trajectory indicators alongside the single-snapshot figures already reported by existing tooling.

CONCLUSION

Bus-factor analysis has matured considerably as an estimation problem; however, it almost universally reports its result as a single number computed at a single point in time. This paper has proposed treating the bus factor as a trajectory instead, defined three summary statistics for characterizing that trajectory, and rather than only proposing the approach, applied it to the complete commit history of three real, actively maintained packages. The measured results show that packages with an identical current bus factor can have markedly different long-run patterns of maintenance concentration, which is precisely the distinction a single snapshot cannot make. A concrete protocol for extending this pilot to a larger, cross-registry sample is specified as the direct next step, and the pipeline used here requires no structural change to be applied at that scale.

REFERENCES

1. Cloud Native Computing Foundation. (2024). XZ Utils compromise [Online]. CNCF TAG Security. Available from: <https://tag-security.cncf.io/community/catalog/compromises/2024/xz/>.
2. United States. Superintendent of Documents. National Institute of Standards and Technology. US Government Printing Office; 1992.
3. Avelino G, Passos LT, Hora AC, Valente MT. A novel approach for estimating Truck Factors. 24th IEEE International Conference on Program Comprehension (ICPC), Austin, TX, USA. 2016. p.1–10. doi:10.1109/ICPC.2016.7503718.
4. Zazworka N, Stapel K, Knauss E, Shull F, Basili VR, Schneider K. Are developers complying with the process: an XP study. 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Bolzano-Bozen, Italy. 2010. p.1–10. doi:10.1145/1852786.1852805.
5. Hannebauer C, Gruhn V. Algorithmic complexity of the truck factor calculation. Product-Focused Software Process Improvement: 15th International Conference, PROFES 2014, Helsinki, Finland. 2014. p.119–33. doi:10.1007/978-3-319-13835-0_9.
6. Cosentino V, Cánovas Izquierdo JL, Cabot J. Assessing the bus factor of Git repositories. 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, Canada. 2015. p.499–503. doi:10.1109/SANER.2015.7081864.
7. Ferreira MM, Valente MT, Ferreira KAM. A comparison of three algorithms for computing truck factors. 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), Buenos Aires, Argentina. 2017. p.207–17. doi:10.1109/ICPC.2017.35.
8. Jabrayilzade E, Evtikhiev M, Tüzün E, Kovalenko V. Bus factor in practice. 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Pittsburgh, PA, USA. 2022. p.97–106. doi:10.1145/3510457.3513082.
9. Piccolo SA, De Meo P, Terracina G. Evaluating and improving projects' bus-factor: a network analytical framework. In: Social Networks Analysis and Mining: 16th International Conference, ASONAM 2024, Rende, Italy. 2024. p.255–70. doi:10.1007/978-3-031-78541-2_16.
10. Haratian V, Evtikhiev M, Derakhshanfar P, Tüzün E, Kovalenko V. BFSig: leveraging file significance in bus factor estimation. 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), San Francisco, CA, USA. 2023. p.1926–36. doi:10.1145/3611643.3613877.
11. Almarimi N, Ouni A, Chouchen M, Mkaouer MW. Improving the detection of community smells through socio-technical and sentiment analysis. *J Softw Evol Process*. 2023;35(6):e2505. doi:10.1002/smr.2505.
12. Rigby PC, Zhu YC, Donadelli SM, Mockus A. Quantifying and mitigating turnover-induced knowledge loss: case studies of Chrome and a project at Avaya. 38th International Conference on Software Engineering (ICSE), Austin, TX, USA. 2016. p.1006–16. doi:10.1145/2884781.2884851.