Same Volatility, Different Domains: A Reproduction of CodeLL Across ML and Software Engineering Projects

Gabriel de Souza Mendes
Federal University of Ceará
Fortaleza, CE, Brazil
gabrielmendes01@alu.ufc.br

Francisco Paulino Arruda Filho
Federal University of Ceará
Fortaleza, CE, Brazil
paulinofilho@alu.ufc.br

Camilo Almendra
Federal University of Ceará
Fortaleza, CE, Brazil
camilo.almendra@ufc.br

Lincoln Souza Rocha
Federal University of Ceará
Fortaleza, CE, Brazil
lincoln@dc.ufc.br

Abstract

Code change datasets rarely capture the long-term temporal dimension of repositories, which limits their use in lifelong learning scenarios for code language models. The CodeLL dataset fills this gap by mapping changes in files, methods, and API calls across the entire release history of 71 Python machine learning (ML) projects. The released dataset, however, ships only descriptive analyses without aggregated churn metrics, covers a single application domain, and its target repositories have continued to evolve since the original mining in 2023/24. This paper addresses these gaps through a reproduction and extension of the CodeLL study: we re-extracted six projects from Software Heritage, three ML projects and three software engineering (SE) tool projects, totaling 615 releases, and re-executed the original mining pipeline and statistics notebook on the newly mined data. The pipeline ran end to end on all six projects, and for Gensim—the only project allowing a direct comparison against the published figures—the descriptive patterns were reproduced, with coefficient deviations compatible with repository growth between the two mining dates. Our extension adds two churn metrics for API calls and methods, plus two derived analyses (a volatility ranking and a per-release API change rate). The cross-domain comparison showed that this first operationalization does not distinguish the six sampled projects: its value ranges are nearly identical in ML and SE. Codebase size and maturity are a candidate explanation, but the available size proxy confirms it only in part, so it remains a hypothesis. We further document qualitative evidence of real, datable API evolution (Python 2→3 migration) in both domains, and report the practical difficulties of the reproduction, notably the rate limiting of Software Heritage.

Keywords

Research reproduction, Software evolution, Mining software repositories, API evolution, Lifelong learning, CodeLL

1 Introduction

Language models for code (code LMs) are typically trained and evaluated as static entities, fitted to a fixed snapshot of a repository. Datasets widely used for pre-training and benchmarking, such as The Pile [6], The Stack [9], CodeXGLUE [13], and CodeSearchNet [8], lack a temporal dimension: they do not capture how a project's code and APIs evolve over time. Code change datasets, such as MegaDiff [15] and CCT5 [12], advance in this direction, but cover only two points in time per sample, which makes them unsuitable for studying lifelong learning scenarios, in which a model must accumulate knowledge about a continuously changing repository [7, 20]. To fill this gap, Weyssow et al. [18] proposed the CodeLL dataset, mined from the Software Heritage (SWH) platform [3], which records code changes – at the file, method, and function/API call level – across the entire release history of 71 Python repositories related to machine learning (ML), totaling 2,483 releases and approximately 970,000 **.py** files.

Despite filling the temporal gap, the CodeLL study leaves three open issues. First, its released analyses are purely descriptive: the dataset ships no aggregated churn metrics at the release or API level, even though code churn is a long-established indicator of software evolution and defect proneness [11, 17], and API-level change in particular has direct impact on client code [4, 14]. Second, the corpus covers a single application domain (ML); empirical evidence shows that findings obtained from a single slice of projects do not generalize, and that sample diversity must be an explicit design concern in repository mining studies [16] – which calls for cross-domain comparisons before ML-specific conclusions about API evolution are drawn. Third, the mined repositories have continued to evolve since the original extraction in 2023/24, so the published snapshot no longer reflects the current state of the projects.

The goal of this work is to characterize long-term API and method churn (which we use to assess API volatility) based on the CodeLL data schema and to compare this churn across two software domains: ML projects and SE tool projects. Answering these questions on newly mined data requires, as a precondition, verifying that the CodeLL mining pipeline still runs end to end and yields results consistent with those originally reported. We therefore conducted a reproduction in the ACM sense of the term (i.e., a different team using the original artifacts [1]) by re-executing the CodeLL mining pipeline and the original statistics notebook [19] on six projects re-extracted from Software Heritage in 2026.

The main contributions of this work are: (i) two churn metrics for API calls and methods, plus two derived analyses (a volatility ranking and a per-release API change rate), implemented on top of the CodeLL schema; and (ii) a cross-domain comparison showing that this first operationalization of API volatility does not distinguish the six sampled projects by domain. As secondary contributions, we report (iii) the re-execution of the original artifacts

on six re-extracted projects (615 releases), which ran end to end on all six and reproduced the descriptive findings of the original study for Gensim; and (iv) the practical difficulties of the reproduction, including Software Heritage infrastructure limitations.

2 The Reproduced Study

Our reproduction is built on two artifacts of the CodeLL study [18]: its data schema and its mining pipeline, described below. The dataset was mined from Software Heritage [3], whose deterministic re-extraction [10] lets us re-mine the same projects at a later date, and targeted ML projects because their libraries evolve rapidly [5].

The CodeLL schema is organized around the Archive entity (a repository, identified by an origin URL), which contains multiple Releases (identified by branch and date). Each Release contains SourceCodeFiles (characterized by imports), which in turn contain Methods (with parameters), which contain FunctionCalls (with start and end positions in the code). Files, methods, and calls are specializations of a common entity, MappedEntity, identified by a unique hash and a mapping_type attribute that indicates whether the entity was mapped, added, or removed between two releases. The previous_release association links each SourceCodeFile to its previous version, forming the basis of all code change analysis.

The CodeLL mining pipeline comprises five stages: (1) collecting the releases via the SWH API; (2) filtering out pre-releases; (3) downloading the source code of each release; (4) static extraction of the **.py** files; and (5) generating the **.jsonl** files with the changes mapped between consecutive releases. The mapping between two releases r_m and r_n follows three heuristics: files are mapped by identical relative path or, in case of refactoring, by Levenshtein similarity ≥ 0.75 between names combined with the presence of identical functions/methods; methods are mapped by belonging to the same class and having the same name and parameters; and function/API calls are mapped by the same code position, with ties broken by positional proximity. Unmapped elements of r_m are marked as removed, and unmapped elements of r_n as added.

Two inherited properties condition how our metrics must be read. First, what the pipeline extracts—and what we call an API call here—is any function call inside a parsed method, with no distinction between internal calls, built-ins, standard-library, third-party, or public-interface calls. Second, entity identity is content-based: a call is identified by its expression and its offsets *within the enclosing method*, a method by its class name, name, and parameters; neither includes the file path.

3 Related Work

Our reproduction builds on prior work along three axes: the infrastructure for reproducible repository mining, the corpora that motivate temporally-aware datasets for code LMs, and the empirical literature on churn and API evolution against which our metrics should be positioned.

Di Cosmo and Zacchiroli [3] present Software Heritage, an initiative to collect and preserve all publicly available source code in an append-only archive built on a Merkle DAG, with intrinsic, hash-based identifiers ensuring artifact integrity. This immutability is precisely the property our reproduction relies on to re-mine the same projects two years after the original study. Building on that archive, Lefeuvre et al. [10] propose fingerprints—a query paired with a timestamp—that translate OCL queries into deterministic extraction programs over SWH, reporting 96.8% accuracy in reconstructing datasets over time; this deterministic re-extraction is the mechanism that lets a reproduction target the exact projects mined by CodeLL. On the corpus side, Gao et al. [6] introduce The Pile, an 825 GiB corpus of 22 diverse subsets built on the premise that data diversity improves generalization. Corpora such as this underpin modern code LMs, yet they capture only static snapshots—the temporal gap that CodeLL [18], and by extension this reproduction, sets out to address.

Neither the churn metrics nor the volatility analyses we add are new as concepts. Nagappan and Ball [17] established relative churn—normalized by a size denominator such as LOC—as a defect predictor; our rate follows the same intuition, but its denominator is the call events in the release rather than a size measure, a choice Section 5.3 shows to be consequential. Dig and Johnson [4] classify API changes by the refactorings causing them, whereas our ranking is purely frequency-based; McDonnell et al. [14] measure stability over explicitly delimited public APIs, a delimitation we lack. Our contribution is thus not the measures but their operationalization over the CodeLL schema.

4 Reproduction Methodology

To guide our investigation, we adopted the Goal-Question-Metric approach [2], defining our objective as follows: to analyze the CodeLL mining pipeline and the data it produces; for the purpose of reproducing the original findings and extending them with churn metrics and a cross-domain comparison, with respect to code and API evolution across releases; from the point of view of software evolution researchers, in the context of six Python projects from two application domains. This objective unfolds into three research questions (RQs), detailed below.

RQ1: Does re-executing the CodeLL mining pipeline on newly extracted data reproduce the descriptive findings reported in the original study? – RQ1 establishes the reproduction itself: it verifies that the publicly released artifacts still run end to end and that the descriptive analyses they produce remain consistent with those reported in the original study, which is a precondition for trusting any extension built on top of them.

RQ2: What evolution patterns do churn metrics for API calls and methods reveal beyond the analyses of the original study? – RQ2 investigates whether aggregated churn metrics – absent from the released analyses, despite churn being a classic indicator of software evolution [11, 17] and API change having direct client impact [4, 14] – surface evolution signals not visible in the original descriptive statistics.

RQ3: Do ML projects exhibit higher API volatility than SE tool projects? – RQ3 tests, cross-domain, a natural extrapolation of the original study's motivation that ML libraries evolve rapidly [5]. Since findings obtained from a single project slice do not generalize [16], RQ3 contrasts the ML domain against a comparison group from a different domain.

4.1 Project Selection and Data Extraction

We performed the reproduction using artifacts publicly available from the CodeLL authors [19], with the original mining pipeline and statistics notebook. However, instead of re-mining the 71 machine learning projects from the original corpus, we selected six projects distributed across two domains. This choice was a deliberate design decision for two reasons. First, verifying RQ3 required cross-domain comparisons using 100% Python projects with long release histories—an intentional sampling criterion that most of the original corpus does not necessarily meet. Second, the SWH API imposes hourly request quotas, recognized by the CodeLL authors themselves as the main bottleneck in the mining process [18], which increases the mining cost with each additional project. Furthermore, while generating an authentication token increases the quota, the restriction still limits the number of projects that can be mined within a fixed request budget.

We selected three projects from the machine learning domain (gensim, spacy, lightgbm), where gensim and lightgbm belong to the original CodeLL corpus, and three projects from the software engineering tools domain (black, mypy, pytest), used as a comparison group. Two criteria were applied: the project had to be written entirely in Python, since the CodeLL parser only processes **.py** files, and it had to belong to one of the two target domains; all selected projects also satisfy CodeLL's minimum requirement of at least two releases per repository. This is a purposive convenience sample: we defined no popularity threshold, age criterion, or enumerated candidate pool, and the groups were not matched by size, age, or number of releases, nor characterized along dimensions such as project role. Section 5.3 interprets the cross-domain results in light of this design. Data extraction from SWH was performed in the week of June 22, 2026; therefore, all numbers reported in this study reflect the state of the repositories on that date, while the original dataset reflects their state at the original mining in 2023/24. Table 1 summarizes both snapshots. In total, our ML slice comprised 254 versions and 1,832 unique files, and the SE slice comprised 361 versions and 1,516 unique files. The entire dataset is 100% Python.

Table 1: Projects, releases, and unique .py files in the original CodeLL dataset and in our reproduction. SE projects and spacy are not part of the original dataset (–). Original mining: 2023/24; our mining: week of June 22, 2026. Release counts are those retrieved by the pipeline; the notebooks apply further filters (Section 4.2).

		CodeLL (2023/24)		Ours (2026)	
Project	**Dom.**	**Rel.**	**Files**	**Rel.**	**Files**
gensim	ML	77	263	79	263
lightgbm	ML	17	14	19	13
spacy	ML	–	–	156	1,556
black	SE	–	–	35	479
mypy	SE	–	–	137	617
pytest	SE	–	–	189	420
Total	–			**615**	**3,348**

4.2 Reproduction and Extension Procedures

The analyses were conducted in two notebooks. To answer RQ1, we ran the original statistics notebook, without modifications, on the **.jsonl** files produced by re-executing the CodeLL mining pipeline on our six projects; keeping the original artifacts untouched cleanly separates what is reproduction from what is extension. To answer RQ2 and RQ3, we implemented an extended notebook over the same **.jsonl** files, providing two churn metrics plus two derived analyses.

The two churn metrics are: the API churn per release, defined as the proportion of function/API calls added, removed, changed, and unchanged at each release, with the average Levenshtein distance of the changed entities overlaid as a second dimension; and the method churn per release, which applies the same logic at the method level. An entity is *changed* when it was mapped to a counterpart in the previous release with a Levenshtein distance greater than zero, and *unchanged* when mapped with distance zero; *added* and *removed* follow the mapping_type assigned by the pipeline.

The two derived analyses are the ranking of most volatile APIs, which orders calls by name according to the addition and removal events accumulated over the project history, and the per-release API change rate, the measure used to compare projects and domains in RQ3. For a project p with releases $r_1, \ldots, r_N$ ordered by date, let a_i, d_i, c_i, and u_i denote the call events added, removed, changed, and unchanged at release r_i. The rate is then

$$\rho_i = \frac{a_i + d_i + c_i}{a_i + d_i + c_i + u_i}, \qquad V_p = \frac{1}{N} \sum_{i=1}^{N} \rho_i. \tag{1}$$

The denominator of ρ_i is the number of call events in *that release*, not a project-level total, and V_p is an *unweighted* mean over releases; no release is excluded from the sum, a consequence discussed in Section 6.

The notebooks filter releases differently, which matters when reading the figures. The original one keeps pre-releases, groups records by branch name, and discards each project's first release, which has no predecessor. The extended one removes pre-releases—branches containing `.dev`, `rc`, `alpha`, `beta`, `.a`, or `.b`—orders releases by date, and retains the first. The counts plotted are therefore smaller than those in Table 1.

5 Results and Discussion

Below, we present the results organized by research question. Each subsection begins with the direct answer to the research question, followed by the analyses and examples that support it.

5.1 RQ1: Does re-executing the CodeLL mining pipeline on newly extracted data reproduce the descriptive findings reported in the original study?

The pipeline and the original statistics notebook ran end to end on all six re-extracted projects. For Gensim—the only project in our sample for which a direct comparison against the published figures is possible—the descriptive patterns of the original study were reproduced; the only deviations observed are quantitative and

are consistent with the growth of the repository between the two mining dates.

We consider an analysis reproduced when the re-executed notebook yields the same qualitative pattern as the published figure—same shape, same dominance among categories, same direction of association—admitting numerical deviations attributable to the difference between snapshots. We did not attempt the original aggregate statistics over 71 repositories, which our sample cannot support. Only gensim and lightgbm belong to that corpus, and only gensim is detailed figure-by-figure; the other five projects attest that the artifacts still execute, not that their outputs match published values.

The file change analysis (Figure 1) reproduces the qualitative patterns of the original study. Its horizontal axis warrants two clarifications, both inherited from the unmodified original artifact. Of the 79 releases mined for Gensim, the notebook discards the first, leaving 78 plotted; the tick labels stop at 77 because their positions are hard-coded and were calibrated for the 77 releases of the 2023/24 snapshot. Records are also grouped by branch name, so releases are ordered lexicographically, and the axis should be read as an ordered sequence rather than a timeline.

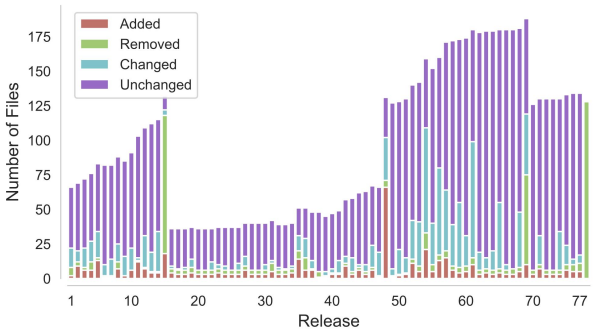

Figure 1: File changes per release of Gensim, obtained by re-executing the original statistics notebook on the data mined in 2026. The axis is an ordered sequence of releases, not a timeline (Section 5.1).

The main deviation appears in the correlation between changed files and changed methods per release (Figure 2). The approximately linear relationship reproduces, but the regression coefficients differ: the original study reports $y = 4.87x - 8.12$, whereas our 2026 snapshot of Gensim yields $y = 5.42x - 15.16$. This is compatible with the releases accumulated between the original mining (2023/24) and ours (June 2026), and does not affect the qualitative pattern; we did not, however, isolate repository growth as the cause, which would require refitting the model on the releases already present in the 2023/24 snapshot. The aggregate distributions also behaved as expected: the distribution of **.py** files per release keeps its long-tail shape even at reduced scale, while the distribution of releases per repository becomes degenerate with only six projects, relative to the smooth distribution obtained over the original 71 repositories.

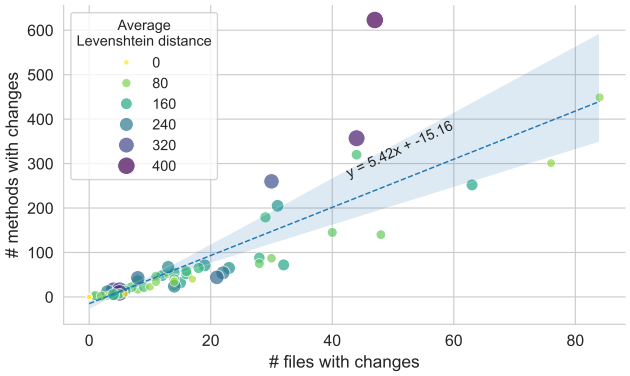

Figure 2: Correlation between changed files and changed methods per release of Gensim, with the average Levenshtein distance as the color dimension, obtained with the original statistics notebook. The in-plot annotation reads $y = 5.42x - 15.16$.

5.2 RQ2: What evolution patterns do churn metrics for API calls and methods reveal beyond the analyses of the original study?

The churn metrics reveal three patterns not visible in the original analyses: (i) in the two projects we inspect in detail, change activity grows as the project matures; (ii) the unchanged category dominates each release, at around 80% of entities in these projects, compressing the change signal and calling for trend-focused visualizations; and (iii) the volatility ranking surfaces real, datable API evolution – such as the Python 2⟶3 migration – in both domains.

Figures 3 and 4 show the churn per release for Gensim (ML, at the method level) and pytest (SE, at the API-call level), illustrating the two granularities of the metric. In both, change activity tends to grow as the project matures and its codebase expands. These are illustrative cases rather than a domain-level finding: the plots are at different granularities, and we computed no per-project summary or trend coefficient across the six projects. In these stacked charts, the unchanged category visually dominates, compressing the added, removed, and changed bands, which are precisely the activities that illustrate the evolution of APIs. We therefore recommend that readings of this type of chart focus on the trend of the change categories over time, or present them on a separate scale, rather than on their absolute values on the chart's scale.

Inspecting the ranking of most volatile APIs of each project (Figures 5 and 6), we found real, datable traces of the evolution of the Python language itself, present in both domains. In Gensim (ML), we identified calls such as **iteritems** and **xrange** – patterns characteristic of Python 2 disappearing over the release history. In pytest (SE), we observed the same phenomenon with **text_type** and string literals prefixed with **u"..."**, typical of the Python 2/3 compatibility layer. Two caveats bound how the ranking should be read. First, it counts absolute change events without normalizing by how often a call occurs, so its top positions hold high-frequency calls such as **len**, **str**, and **isinstance**; the migration markers appear lower, and what the ranking supports is their presence, not their

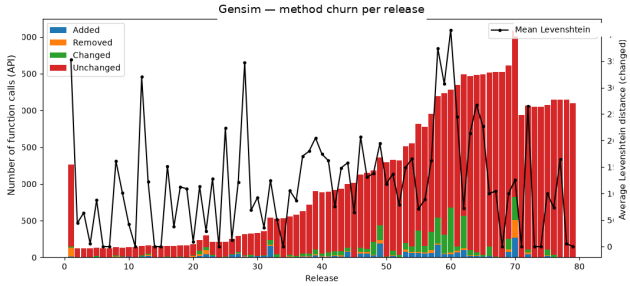

Figure 3: Method-level churn per release of Gensim (ML domain), with the average Levenshtein distance overlaid. The y-axis label is inherited from the shared plotting routine and should read *number of methods* here.

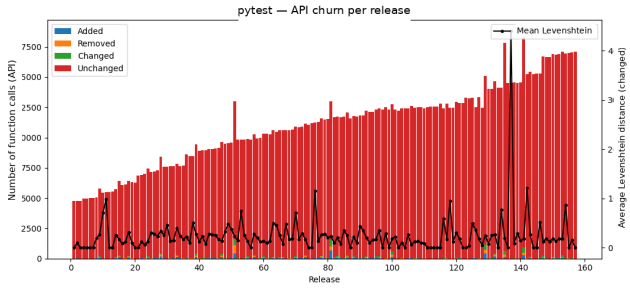

Figure 4: API-call churn per release of pytest (SE domain), with the average Levenshtein distance overlaid. Pre-releases are filtered out, hence fewer releases than in Table 1.

rank. The test-related calls in Figure 5 (**assertEqual**, **testfile**) also show concretely that test and production code are not separated. Second, we did not locate the releases at which these calls disappear, so the evidence is datable in principle but not dated here. With these caveats, the result still reinforces that the pipeline captures the kind of distribution shift that motivates the dataset.

5.3 RQ3: Do ML projects show higher API volatility than SE tool projects?

Under the per-release API change rate of Equation 1, the two domains show nearly identical value ranges (≈ 0.0085 to 0.037); this first operationalization of API volatility does not distinguish the six sampled projects by domain.

We computed the mean rate for the three projects in each domain (Figures 7 and 8). The values obtained were: **lightgbm** ≈ 0.037, **gensim** ≈ 0.020, and **spacy** ≈ 0.010 in the ML domain. In the SE domain we had **black** ≈ 0.037, **mypy** ≈ 0.0135, and **pytest** ≈ 0.0085. The two ranges overlap almost exactly, and the highest value in each domain is virtually identical. The direct comparison thus does not separate ML from SE.

The most plausible explanation is that the rate tracks codebase size and maturity rather than domain: in a small or recent project each change is a larger fraction of the calls observed in a release, whereas in a large one it is diluted. Our data support this only in part. The only size proxy we measured is the number of unique **.py**

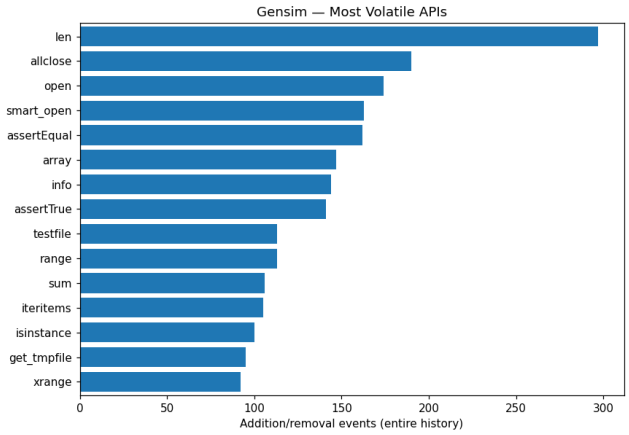

Figure 5: Calls most frequently added or removed over the history of Gensim (ML). The Python 2 markers iteritems and xrange appear among them.

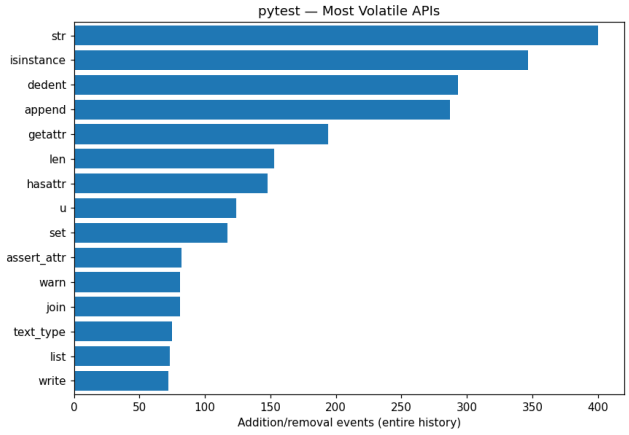

Figure 6: Calls most frequently added or removed over the history of pytest (SE). The Python 2/3 compatibility markers text_type and u"..." appear among them.

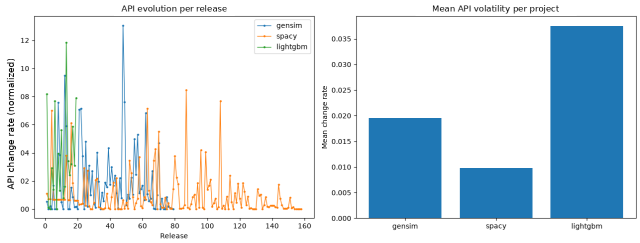

Figure 7: Per-release API change rate of the ML domain projects (gensim, spacy, lightgbm): per release (left) and project mean (right).

files (Table 1). In ML it is consistent, rates decreasing monotonically as file counts grow (lightgbm, 13 files, 0.037; gensim, 263, 0.020;

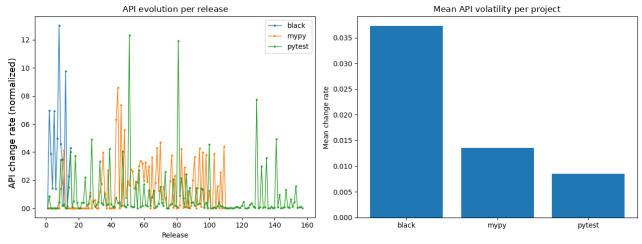

Figure 8: Per-release API change rate of the SE domain projects (black, mypy, pytest): per release (left) and project mean (right).

spacy, 1,556, 0.010). In SE it is not: **pytest** has fewer files than **black** (420 against 479) yet the lowest rate (0.0085 against 0.037), and **mypy**, the largest, sits between them. Having measured neither LOC, total calls, nor project age, we leave size and maturity as a hypothesis.

What holds without qualification is the negative result: the hypothesis that ML projects are systematically more volatile than SE tool projects—a natural extrapolation of the original study's motivation—is not supported by this measure on these six projects, and whether the effect is absent or merely masked cannot be settled with the present design. For future comparisons using CodeLL, we recommend a size denominator external to the release, such as LOC or number of methods, and pairing projects by maturity and size rather than by domain alone.

6 Threats to Validity and Reproduction Difficulties

The main practical difficulty was the SWH API rate limit, mitigable with an authentication token. Parts of the CodeLL artifact repository were also incomplete or poorly documented: running the pipeline end to end required managing the environment with **uv** instead of **pip**, installing **swh-core** via **uv tool**, fixing a mixed absolute/relative import in the code change generation script, and reconciling a script name in the documentation with its actual path. All are environment and import concerns; none alters the extracted data.

Construct validity. The rate of Equation 1 is ours; its denominator is internal to the release rather than a size measure (Section 5.3). The ranking counts absolute events without normalizing by usage frequency. More fundamentally, we use function calls as a proxy for APIs without distinguishing internal, built-in, standard-library, third-party, or public-interface calls, and without separating test from production code; in a project such as pytest this is far from cosmetic.

Internal validity. Beyond the mapping errors under complex refactorings acknowledged in the original study, two issues stem from entity identity (Section 2): identifiers exclude the file path, so distinct entities can collide—our notebook deduplicates per release and resolves collisions in favor of the mapped category, biasing counts towards *unchanged* and depressing the rate by an unquantified amount—and only calls inside parsed methods are recorded, so module-level calls are absent. We also correct a limitation we initially misdiagnosed: each repository's first release, having no predecessor, is written without mapping fields. The original notebook discards it; our extended notebook counts every entity in it as *unchanged*, giving that release a rate of exactly zero. Rather than an inflation, this scales the mean down by $(N-1)/N$—a relative underestimate of roughly $1/N$, under six percent even for the smallest project. Ordering and the RQ3 conclusion are unaffected.

External validity. With three purposively selected Python projects per domain (Section 4.1), our results do not generalize to other ecosystems, and the cross-domain comparison is initial, not conclusive, evidence.

Reproducibility of this study. The **.jsonl** files of our June 2026 extraction were not preserved, which prevents recomputing the derived measures without re-mining and is why the refit of Section 5.1 appears as future work: in longitudinal mining studies, the intermediate representation is part of the artifact.

7 Conclusion and Future Work

This work reported a reproduction and extension of the CodeLL study [18] on six Python projects re-extracted from Software Heritage. The pipeline and the original notebook ran end to end on all six projects, and for Gensim the descriptive patterns were reproduced, with coefficient deviations compatible with repository growth between the mining dates (RQ1). The churn metrics revealed growing change activity in the projects inspected, the visual dominance of the unchanged category, and qualitative evidence of the Python 2→3 migration in both domains (RQ2). A first operationalization of API volatility did not distinguish ML from SE tool projects; codebase size and maturity are a plausible explanation, but one the available size proxy confirms only in part (RQ3).

The Python 2→3 traces speak to what motivates CodeLL: a call such as **iteritems** is a legitimate token at one point of a repository's history and an error at another, a transition no single-snapshot model can represent. That the pipeline recovers such transitions is, in our view, the strongest argument for the dataset. Two lessons stand out: the choice of metric matters more than the volume of data, since a denominator that ignores codebase size can obscure the intended effect; and infrastructure limits such as SWH quotas should be planned for from the outset.

As future work, we plan the following: (i) normalizing volatility by codebase size and pairing projects by maturity, measuring codebase size (LOC, methods) and project age directly rather than hypothesizing them as in Section 5.3; (ii) refitting the changed-files versus changed-methods regression of Section 5.1 on the releases in the 2023/24 snapshot; (iii) making entity identity location-sensitive and stratifying calls by API type and test versus production code; (iv) including more projects per domain; and (v) churn visualizations on a scale not dominated by the unchanged category.

Artifact Availability

The original CodeLL artifacts are available in the authors' repository [19]. We forked it to implement the fixes of Section 6 and add a notebook with the metrics of Section 4.2; these extended artifacts are at https://github.com/gabrielsouzam/CodeLL-Dataset-Reproduction, archived at https://doi.org/10.5281/zenodo.21873097. The **.jsonl** files of the June 2026 extraction are not included.

References

[1] Association for Computing Machinery. 2020. Artifact Review and Badging — Current (Version 1.1). https://www.acm.org/publications/policies/artifact-review-and-badging-current. Accessed: 2026-07-10.

[2] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. Goal Question Metric Paradigm. In *Encyclopedia of Software Engineering*, John J. Marciniak (Ed.). Vol. 1. John Wiley & Sons, New York, NY, 528–532.

[3] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *Proceedings of the 14th International Conference on Digital Preservation (iPRES)*. Kyoto, Japan.

[4] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (2006), 83–107. doi:10.1002/smr.328

[5] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Transactions on Software Engineering and Methodology* 30, 4 (2021), 55:1–55:42. doi:10.1145/3453478

[6] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027* (2020).

[7] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. 30–42. doi:10.1109/ICSE48619.2023.00015

[8] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436* (2019).

[9] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of Permissively Licensed Source Code. *arXiv preprint arXiv:2211.15533* (2022).

[10] Romain Lefeuvre, Jessie Galasso, Benoit Combemale, Houari Sahraoui, and Stefano Zacchiroli. 2023. Fingerprinting and Building Large Reproducible Datasets. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability (ACM REP '23)*. ACM, New York, NY, USA, 27–36. doi:10.1145/3589806.3600043

[11] Meir M. Lehman. 1980. Programs, Life Cycles, and Laws of Software Evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076. doi:10.1109/PROC.1980.11805

[12] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-Trained Model. *arXiv preprint arXiv:2305.10785* (2023).

[13] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021).

[14] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 70–79. doi:10.1109/ICSM.2013.18

[15] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A Dataset of 600k Java Source Code Changes Categorized by Diff Size. *arXiv preprint arXiv:2108.04631* (2021).

[16] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 466–476. doi:10.1145/2491411.2491415

[17] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 284–292. doi:10.1145/1062455.1062514

[18] Martin Weyssow, Claudio Di Sipio, Davide Di Ruscio, and Houari Sahraoui. 2024. CodeLL: A Lifelong Learning Dataset to Support the Co-Evolution of Data and Language Models of Code. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR '24)*. ACM, Lisbon, Portugal, 637–641. doi:10.1145/3643991.3644864

[19] Martin Weyssow, Claudio Di Sipio, Davide Di Ruscio, and Houari Sahraoui. 2024. CodeLL-Dataset: source code and artifact repository. https://github.com/martin-wey/CodeLL-Dataset. Accessed: July 2026.

[20] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. On the Usage of Continual Learning for Out-of-Distribution Generalization in Pre-trained Language Models of Code. *arXiv preprint arXiv:2305.04106* (2023). doi:10.48550/arXiv.2305.04106