The Ground Is Shifting: A Reflection on the Foundations of Software Measurement

Thomas Bock
Carnegie Mellon University
Pittsburgh, PA, USA
bockthom@cmu.edu

Audris Mockus
University of Tennessee
Knoxville, TN, USA
audris@utk.edu

Bogdan Vasilescu
Carnegie Mellon University
Pittsburgh, PA, USA
vasilescu@cmu.edu

Abstract

For most of the past six decades, software measurement relied on labor-intensive manual collection of proprietary data, which hampered progress. The shift to repurposing traces from version control and related tools dramatically expanded data availability—especially with the rise of open-source software—but hinged on an often unstated assumption: that these tools are used by professional developers to build genuine software systems. However, as trace-generating tools, data types and scale, and empirical methods have all evolved, it has become clear that changes in data generation and analytical approaches affect many prior findings about software development, maintenance, and evolution. With AI agents now actively using these same tools, the resulting traces frequently violate the original assumption of human origin. To preserve the relevance of software measurement research, immediate action is needed: We must detect when foundational assumptions are violated in contemporary data and develop new methodologies that remain valid under changed circumstances. To this end, we propose a systematic AI-assisted replication program that revisits key findings using modern techniques, aiming for methods that yield consistent results on current data to keep software measurement meaningful.

CCS Concepts

• **Software and its engineering → Software version control**.

Keywords

Mining software repositories, squash-merging, AI-generated code

ACM Reference Format:
Thomas Bock, Audris Mockus, and Bogdan Vasilescu. 2026. The Ground Is Shifting: A Reflection on the Foundations of Software Measurement. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3832783.3834553

1 Introduction

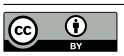
Empirical software engineering has transformed significantly over the past six decades. The initial notion that software should be measured to inform development became widespread in the 1970s, notably with the introduction of source-code complexity metrics [52], regression models for developer productivity [84], and early software reliability models [59]. These foundational approaches, though impactful, relied on costly manual data collection. The 1980s brought more sophisticated models [3, 12], along with systematic programs for software measurement in enterprises [33] and critical evaluations of emerging methodologies [88]. Manual data collection remained prevalent and access to data was mostly proprietary.

In the 1990s, empirical software measurement matured further. Researchers proposed frameworks for effective measurement systems [6, 72], practical guidance for their application [66], and methodologies for qualitative studies [73]. However, measurement continued to be restricted to organizations capable of supporting extensive programs, limiting broader research participation.

A pivotal shift occurred with the realization that valid metrics could be extracted from projects that use version control and issue trackers. This democratized empirical software engineering: Any project using these systems could analyze its own data and benchmark against historical states, enabling predictions about defect introduction [57] and future file reliability [35]. Crucially, validation extended beyond proprietary projects [54, 55], opening vast public open-source datasets to the research community. Methodological advances replaced rules for manual metric collection [45] with automated extraction techniques from operational systems. As platforms evolved—from SourceForge [39] to git's dominance [7] and GitHub's ascendance [42]—data extraction approaches were continually refined. Researchers adapted their tools, methods, and datasets in response to rapid technological change and shifting development practices [28, 64, 78]. The rise of open-source hosting platforms—especially GitHub—dramatically expanded research possibilities. Data could be enhanced by incorporating issue trackers, pull requests (PRs), and continuous-integration logs [15, 16]. This enabled analyses at ecosystem scale rather than isolated projects [13, 53], driving the development of mining infrastructure such as GHTorrent or Software Heritage archives [17, 23, 32, 65].

Recently, datasets have grown to span entire ecosystems. Repository archives such as World of Code aggregate version control data across diverse hosting platforms, constructing unified graphs linking developers, commits, and projects across observable open-source activity [51, 65]. These cross-platform resources allow researchers to address previously intractable questions, such as studying vulnerability propagation via dependencies or examining the diffusion of practices across programming communities [2, 8, 36, 50, 58, 71]. The scale now encompasses billions of artifacts, requiring advanced computational infrastructure and analytics [13, 51, 65].

Methodological rigor has kept pace: Extracting valid measurements from operational systems is now standard practice, with greater attention paid to threats to validity, replication standards, and statistical robustness [5, 27, 47, 77, 83]. Machine learning and natural-language processing techniques are widely adopted—for example, in code completion or vulnerability detection [24, 49].



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834553

Qualitative approaches (e.g., interviews, ethnographies) complement or replace quantitative analyses in many settings [21, 25, 74].

The emergence of large language models (LLMs) and AI-powered development tools marks a new inflection point. Tools such as ChatGPT, GitHub Copilot, or Claude Code are reshaping how developers write, review, and debug code [14, 22, 29, 62, 87]. This evolution challenges established empirical research paradigms: datasets built from human-written code may not reflect current practices; studies validated on pre-AI corpora face questions about ongoing relevance. But also new opportunities arise around understanding developer–AI interaction, measuring the quality impact of generated code, and distinguishing human versus machine contributions. The definition of a "software artifact" worthy of study is itself being renegotiated.

Fundamentally, most of the present empirical work is predicated on repurposing operational data (e.g., issue tracking or version control) to measure phenomena such as developer effort or software quality. As processes and tools evolve—particularly with generative AI (GenAI)—the durability of existing measurement methods is uncertain. Methodologies must account for both professional human activity (e.g., keeping logical changes separate in version control) and deviations (e.g., agent-generated commits). Much effort was devoted to cleaning operational data, such as excluding agent activity [20] or untangling complex commit histories [37]. In the age of GenAI agents, extracting valid data requires major methodological innovation. Recent project practices emphasizing simplified history (e.g., squashed commits) further complicate accurate authorship and evolution tracking. At the same time, some problems with cleaning data may no longer be relevant, for example, when LLMs generate accurate, standardized commit messages [89, 92, 94].

Urgent action is needed to ensure the survival of software measurement as a discipline that repurposes operational data in software projects to gain useful software-engineering insights. It requires intervention from tool makers to generate traces that identify automated and manual activities, and from researchers using these traces to find ways to treat human actions (constrained by physics and biology) and agent actions (not having such constraints).

2 Assumptions of Data Collection Practices and the Evolution of Analytical Methods

The evolution of empirical software-engineering research can be understood through two interrelated dimensions: data sources and analytical methods, each of which has undergone substantial change.

Throughout most of its history, software-engineering measurement was hampered by expensive (and limited) manual data collection and proprietary datasets. Only during the last quarter century, the discovery that data from software tools, such as version control systems, could be used to produce valid measures led to exponential expansion in measurement methods, and, in the case of open-source software, in public datasets of millions of projects [26]. That leap to rely on tool-trace data (i.e., records left by tool usage) required many assumptions—some forgotten by now (see Table 1). Even under these assumptions, trace data had to be augmented for different types of software changes, outliers identified and removed, missing data imputed, and many other contingencies addressed [56]. Numerous violations of foundational assumptions are already documented [40]. For example, Rapaport et al. [68] showed that altered version histories—through force pushes or rebasing—have become commonplace. AlMarzouq et al. [4] emphasized issues of data freshness, API limitations, and the difficulty of establishing ground truth.

Table 1: Foundational assumptions and their violations

Assumption	Violation
A commit is an atomic unit of work	Squash-merging collapses PRs into one commit
Traces are generated by human developers	AI agents & bots generate commits
Version history is append-only	Rebasing & force-pushing rewrite history
Author identity is unique and stable	Identities are fragmented & shared accounts exist

While initial regression and software reliability analyses date from the 1970s, most data analysis methods in software engineering were of a simpler kind, such as descriptive statistics and correlations. Gradually, techniques suitable for software data [18], such as negative binomial and logistic regression, survival analysis, mixed-effects models for hierarchical structure, and Bayesian methods for uncertainty quantification increased in popularity [26, 30]. Recently, the field has also begun grappling with causal inference—propensity score matching, instrumental variables, difference-in-differences, and regression discontinuity—recognizing that evaluating interventions without randomized trials requires explicit causal reasoning through directed acyclic graphs [31, 34]. Network analysis operates at multiple scales: social networks from co-commits or pull requests reveal collaboration patterns [10, 61], dependency networks from package managers expose vulnerability propagation [93], and co-change networks illuminate architectural coupling [95]. Machine learning has progressed from traditional defect prediction to deep learning on code representations and LLMs [1, 41, 44, 85], whose integration into workflows (e.g., GitHub Copilot) creates new challenges as code increasingly reflects AI assistance [63].

Sophisticated methods assume stable relationships, but a model validated on repositories in 2010 may not generalize to 2020 data because the phenomenon itself has changed. Zimmermann et al. [96] identified recurrent challenges: establishing ground truth, overgeneralizing from convenience samples, and the tension between statistical and practical significance. Changes in tools alter both how software is developed and recorded, potentially invalidating assumptions underlying data collection and analytical methods [38].

3 Discussion: Which Findings Are Replicable?

The preceding sections documented how data sources and analytical methods have evolved over decades of empirical software engineering research, undermining fundamental assumptions that event traces are generated by a professional human actor following a well-established process with strict quality gates. Would a historic analysis repeated without adjusting for the changes in how the data are now generated lead to the same conclusions? Of the four assumption violations listed in Table 1, we explore two in depth through illustrative anecdotes concerning changes in software development practices, before turning to the broader implications for the field and the potential role of replication and AI in addressing them.

3.1 Squash-Merging Hides Developer Activity

Changes in development practices affect the data empirical researchers analyze. Squash-merging—the practice of collapsing all commits from a pull request into a single commit on the main branch [9, 43]—exemplifies how modern workflows can fundamen-

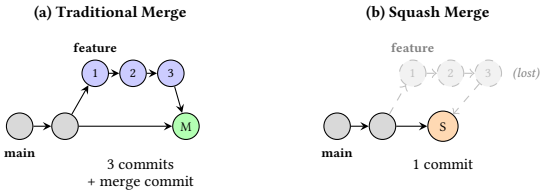


Figure 1: The same pull request results in different histories: (a) traditional merging preserves all commits; (b) squash-merging collapses the commit history into a single commit.

tally alter what repository mining can observe. In Figure 1, we illustrate the difference between traditional merge commits and squash merges. In a traditional merge, the feature branch's entire commit history with every single commit is preserved in the main branch, maintaining a granular record of development activity. In contrast, squash-merging condenses all commits of the pull request into a single commit, erasing intermediate steps. This practice, while beneficial for maintaining a clean main branch history, has significant implications for empirical research. Consider what a researcher analyzing commit histories would see when examining a feature implementation under different merge strategies. With traditional merge commits, the history preserves the development trajectory: initial implementation attempts, debugging iterations, responses to code reviews, and final refinements appear as distinct commits with their own timestamps, authors, and messages. This granular history enables analyses of development processes, bug introduction patterns, and collaboration dynamics. With squash-merging, this trajectory collapses into a single commit. A feature that required three weeks of development, twenty intermediate commits, and input from multiple reviewers appears as one atomic change attributed to a single author on a single date. Debugging sessions disappear. Reviewer-prompted modifications vanish. What remains is a sanitized commit that presents the final outcome as if it emerged fully formed.

The implications for research extend across multiple domains. Bug introduction analysis, which traces defects to the commits that introduced them, cannot pinpoint introductions within squashed commits—the bug and its fix may both be subsumed in the same atomic change [9, 67]. Developer productivity metrics based on commit counts become meaningless when a month of work is squash-merged into a single commit, obscuring actual effort and activity. Similarly, analyses of code churn (i.e., frequency and volume of code changes) are distorted when multiple incremental changes are consolidated into one. Collaboration pattern analysis conducted solely on the project's commit history does not see the changes that might have been conducted during pull-request review. Also, studies examining the temporal dynamics of development lose fidelity when intermediate timestamps are lost. Further, developer networks constructed from co-change relationships between commits become sparser and less informative, neglecting the incremental changes on which the developers have worked. If multiple developers collaborated on a feature branch but their contributions are squashed into a single commit, the resulting network fails to capture the true collaboration structure.

The problem compounds because squash-merging adoption is neither universal nor random. Some projects mandate it; others forbid it; many leave the choice to individual contributors. For

Table 2: Number of commits in merged pull requests (PRs)

Project w. squash-merges	# PRs	Avg. commits/PR	Max. commits/PR	PRs > 1 commits
denoland/deno	15 142	4.3	389	61.8%
electron/electron	25 669	2.8	1 256	38.9%
facebook/react	12 836	2.6	1 900	35.3%
flutter/flutter	48 051	3.6	1 398	48.7%
huggingface/transformers	19 353	7.3	846	64.7%
microsoft/vscode	47 094	2.8	3 896	38.4%
microsoft/TypeScript	14 877	3.8	438	51.7%
twbs/bootstrap	9 259	3.3	9 187	42.7%

Time window: Project start on GitHub until May 12, 2026. Only merged PRs are considered.

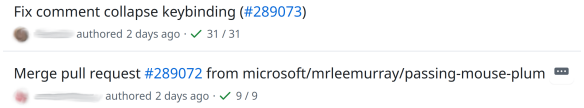


Figure 2: Excerpt of the commit history of microsoft/vscode, showing a traditional merge commit (lower part) directly next to a squash-merged commit (upper part).

example, the commit history of project microsoft/vscode currently shows a mixture of traditional merge commits of pull requests (indicated by *"Merge pull request ..."*) and squash-merged pull requests (indicated by *"(# PR number)"* at the end of the commit message) next to each other, as we show in Figure 2. Analyses that treat commits as comparable units across such heterogeneous data risk systematic biases whose direction and magnitude depend on the correlation between merge practices and the phenomena under study. To illustrate the magnitude of the potential problem, we collected descriptive statistics via the GitHub API in May 2026 for 8 widely-used software projects that (at least, partially) use squash-merging. As we show in Table 2, merged pull requests on average contain 2–7 commits, with maxima up to 9 187 commits. 35%–65% of the pull requests in these projects contain more than one commit and vanish into a single commit when squash-merged. To assess how prevalent squash-merging is at population scale, we then identified all GitHub projects that have at least one commit whose title follows GitHub's standardized pattern for squash-merged pull requests (i.e., the title ends with *"(# PR number)"*). Using data from the repository archive World of Code [51] until October 2025, considering only public projects with more than 1 000 commits (as merging practices primarily matter in projects with large activity), we found that 132 251 out of 341 397 projects (39%) contain at least one such squash-merged commit, indicating that squash-merging is not uncommon. Project-level adoption of squash-merging rose from 18% in 2016 to 39% in 2025, indicating a growing trend. Our data collection even undercounts squash merges, as project owners can customize the format of commit messages for squash merges, making automatic identification of all squash merges even more complicated. To the best of our knowledge, past studies have not sufficiently investigated how their results are dependent on these merge practices.

3.2 Human-Generated vs. AI-Generated Data

Squash-merging already demonstrated that development practices can erase authentic history. Whereas this erasure is a side effect of a workflow choice, an AI agent can create a sequence of commits with realistic timestamps, messages, and diffs, which also does not reflect genuine human development. Even human commits are now

Table 3: Examples of proposed replications targeting human-behavior constructs

Construct	Classic finding	What to replicate	Why it might break
Productivity & effort estimation	Commit count & frequency serve as effort proxy [60, 70, 75, 79]	Analyze commit activity and developer time for pre- and post-GenAI periods	Squash-merged commits as well as AI-generated commits do not reflect development time and human effort
Bug introduction & prediction	Code churn predicts defects [48, 76, 80]	Measure churn in projects with/without squash-merging and with/without AI assistance	Squash-merging hides change frequency; code churn of AI-generated code may reflect insufficient prompting rather than defects
Developer expertise & code ownership	Code contributions reflect expertise [19]	Investigate contributions and their quality	Expertise may shift from code writing to reviewing/prompting

hybrid artifacts—partly human intent, partly AI generation. When a developer uses GitHub Copilot to write a function and then commits it, the trace looks identical to a purely human-authored commit [90], but the underlying generative process is fundamentally different. The distinction between "human" and "artificial" is no longer binary—it is a spectrum that current tooling cannot resolve. With a wider deployment of GenAI agents that use not just version control, but also conduct code reviews, etc., historically held assumptions about humans generating all important software-engineering events no longer hold [69, 82]. At least some software produced entirely via prompts would no longer be of primary interest for human-behavior constructs, but the corresponding prompts might be [86]. Consequently, without a significant change, the current way of software measurement via trace data would no longer be relevant.

3.3 A Call for Action

These anecdotes illustrate a broader concern: classic findings may not be repeatable (without adjustments) on contemporary data that do not satisfy the same assumptions as historical data or if examined using different methods. We do not know how many classic findings survive under contemporary conditions. So, what action may be needed to keep the approach anchored in reality? First, as we exemplify, the assumptions under which software-engineering events are generated are changing. We urgently need to create methods that work under new assumptions (e.g., redefining measurement constructs to account for AI-assisted commits, or developing new metrics that capture human effort in the presence of AI assistance). This includes detecting the types of assumption violations and irregularities listed in Table 1, as well as new ones that yet remain to be identified. Second, we need a broad and systematic replication agenda to re-analyze established constructs using contemporary methods on past and present projects, focused specifically on constructs that measure human behavior. In Table 3, we outline examples of specific replications. Each example targets a finding about human behavior and asks whether it survives when the human-origin assumption is relaxed. Such replications enable *measurement validity research*: determining which constructs still have valid operationalizations and which need entirely new ones. A systematic replication agenda would catalog influential findings, identify ones that could benefit from newer methods or might be affected by new data-generation mechanisms, and subject them to replication with contemporary methods and contemporary data.

The traditional obstacle to replications is cost, as a thorough replication might occupy a graduate student for months. LLMs can alter this calculus. AI assistants such as ChatGPT or Claude can read papers, extract methodological details, and implement analysis pipelines with speed that humans cannot match [46, 91]. An AI-assisted workflow could: (1) ingest papers and extract research questions, methods, and assumptions (e.g., about commit granularity); (2) implement and adapt methodologies for modern data; (3) execute replications at scale across thousands of repositories; (4) synthesize results identifying where findings replicate or fail. Human researchers remain essential to validate implementations, interpret discrepancies, and assess whether adaptations preserve the spirit of original analyses. But AI assistants enable systematic evaluation at a scale that was previously infeasible, as we were limited by human resources. Beyond replication, AI can assist prospectively by identifying assumptions that may be violated by modern practices and suggesting sensitivity analyses. To that end, we need to define units of replication, which findings to re-analyze (e.g., code authorship attribution, productivity, collaboration patterns), and what controls to put in place to ensure that AI replicates the original analysis as faithfully as possible, while adjusting for changes in data generation and methodological advances.

Limitations must be acknowledged: AI may misinterpret methodological details, cannot access proprietary data, and may introduce biases. Therefore, human assessment remains essential [81]. Nevertheless, the combination of accumulated findings, methodological advances, and AI tools creates conditions for comprehensive reassessment of what software-engineering research knows.

Beyond a replication agenda, the research community should advocate that development tools and platform owners (e.g., GitHub, GitLab) generate *provenance metadata* as a first-class part of the commit record: Was this commit human-authored, AI-generated, or AI-assisted? Which AI model was used? What was the prompt? What was the human's review decision? Such metadata would let the field continue its tradition of adapting to new trace-generating technologies, as it has done for six decades, and would enable transparency, traceability, and accountability in software development, fostering trust in the software supply chain and enabling researchers to understand how AI is shaping development practices.

4 Conclusion

Data and methods of empirical software engineering have both transformed over decades—yet we have not systematically examined whether findings established under earlier conditions still hold. Automation using the same tools as human developers makes the assumption that we analyze events of a human actor rapidly obsolete. This reflection is a call for action: The community should prioritize the development of techniques to replicate classic software measurement studies using contemporary methods on contemporary data, leverage AI-assisted tools to make such replication feasible at scale, and treat the results—whether confirmatory or contradictory—as valued contributions that strengthen our collective understanding.

Acknowledgments

We received funding from the National Institutes of Health (NIH) under project no. 1R01GM164731-01, from the Alfred P. Sloan Foundation under grant no. g-2025-25171, and from Schmidt Sciences.

Data Availability Statement

The scripts used to collect the descriptive statistics on squash-merging prevalence presented in Section 3.1, as well as more details and plots showing the adoption of squash-merging over time, are available on Zenodo [11]: https://doi.org/10.5281/zenodo.21727085

References

[1] R. Abreu, V. Murali, P. C. Rigby, C. Maddila, W. Sun, J. Ge, K. Chinniah, A. Mockus, M. Mehta, and N. Nagappan. 2025. Moving faster and reducing risk: Using LLMs in release deployment. In *Proc. Int. Conf. Softw. Eng.: Softw. Eng. in Practice (ICSE-SEIP)*. IEEE, 448–457. doi:10.1109/ICSE-SEIP66354.2025.00045

[2] A. Agroskin, E. Lyulina, S. Titov, and V. Kovalenko. 2023. Constructing temporal networks of OSS programming language ecosystems. In *Int. Conf. Softw. Analysis, Evolution, and Reengineering (SANER)*. IEEE, 663–667. doi:10.1109/SANER56733.2023.00067

[3] A. J. Albrecht and J. E. Gaffney. 1983. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Trans. Softw. Eng. (TSE)* 6 (1983), 639–648. doi:10.1109/TSE.1983.235271

[4] M. AlMarzouq, A. AlZaidan, and J. AlDallal. 2020. Mining GitHub for research and education: Challenges and opportunities. *Int. J. Web Inf. Syst. (IJWIS)* 16, 4 (2020), 451–473. doi:10.1108/IJWIS-03-2020-0016

[5] A. Ampatzoglou, S. Bibi, P. Avgeriou, M. Verbeek, and A. Chatzigeorgiou. 2019. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. *Inf. Softw. Technol. (IST)* 106 (2019), 201–230. doi:10.1016/j.infsof.2018.10.006

[6] V. R. Basili. 1992. *Software modeling and measurement: The Goal/Question/Metric paradigm.* Technical Report. https://www.cs.umd.edu/~basili/publications/technical/T78.pdf

[7] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. 2009. The promises and perils of mining git. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. IEEE, 1–10. doi:10.1109/MSR.2009.5069475

[8] O. A. Blanthorn, C. M. Caine, and E. M. Navarro-López. 2019. Evolution of communities of software: Using tensor decompositions to compare software ecosystems. *Appl. Netw. Sci.* 4, 1 (2019), 120. doi:10.1007/s41109-019-0193-5

[9] P. Bludau and A. Pretschner. 2022. PR-SZZ: How pull requests can support the tracing of defects in software repositories. In *Int. Conf. Softw. Analysis, Evolution, and Reengineering (SANER)*. IEEE, 1–12. doi:10.1109/SANER53432.2022.00012

[10] T. Bock, N. Alznauer, M. Joblin, and S. Apel. 2023. Automatic core-developer identification on GitHub: A validation study. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 32, 6 (2023), 138. doi:10.1145/3593803

[11] T. Bock, A. Mockus, and B. Vasilescu. 2026. *The adoption of squash-merging in GitHub projects – Scripts and descriptive statistics.* doi:10.5281/zenodo.21727085

[12] B. W. Boehm. 1981. *Software engineering economics.* Prentice-Hall.

[13] P. Boldi, A. Pietri, S. Vigna, and S. Zacchiroli. 2020. Ultra-large-scale repository analysis via graph compression. In *Int. Conf. Softw. Analysis, Evolution, and Reengineering (SANER)*. IEEE, 184–194. doi:10.1109/SANER48275.2020.9054827

[14] W. Chatlatanagulchai, K. Thonglek, B. Reid, Y. Kashiwa, P. Leelaprute, A. Rungsawang, B. Manaskasemsak, and H. Iida. 2025. On the use of agentic coding manifests: An empirical study of Claude Code. In *Int. Conf. Product-Focused Softw. Process Improvement (PROFES)*. Springer, 543–551. doi:10.1007/978-3-032-12089-2_40

[15] V. Cosentino, J. L. Cánovas Izquierdo, and J. Cabot. 2017. A systematic mapping study of software development with GitHub. *IEEE Access* 5 (2017), 7173–7192. doi:10.1109/ACCESS.2017.2682323

[16] V. Cosentino, J. Luis, and J. Cabot. 2016. Findings from GitHub: Methods, datasets and limitations. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. ACM, 137–141. doi:10.1145/2901739.2901776

[17] O. Dabić, R. Tufano, and G. Bavota. 2024. SEART data hub: Streamlining large-scale source code mining and pre-processing. In *Proc. Int. Conf. Softw. Maintenance and Evolution (ICSME)*. IEEE, 888–892. doi:10.1109/ICSME58944.2024.00097

[18] F. G. de Oliveira Neto, R. Torkar, R. Feldt, L. Gren, C. A. Furia, and Z. Huang. 2019. Evolution of statistical analysis in empirical software engineering research: Current state and steps forward. *J. Syst. Softw. (JSS)* 156 (2019), 246–267. doi:10.1016/j.jss.2019.07.002

[19] T. Dey, A. Karnauch, and A. Mockus. 2021. Representation of developer expertise in open source software. In *Proc. Int. Conf. Softw. Eng. (ICSE)*. IEEE, 995–1007. doi:10.1109/ICSE43902.2021.00094

[20] T. Dey, S. Mousavi, E. Ponce, T. Fry, B. Vasilescu, A. Filippova, and A. Mockus. 2020. Detecting and characterizing bots that commit code. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. ACM, 209–219. doi:10.1145/3379597.3387478

[21] M. Di Penta and D. A. Tamburri. 2017. Combining quantitative and qualitative studies in empirical software engineering research. In *Companion Volume ICSE*. IEEE, 499–500. doi:10.1109/ICSE-C.2017.163

[22] H. V. F. dos Santos, V. Costa, J. E. Montandon, and M. T. Valente. 2026. Decoding the configuration of AI coding agents: Insights from Claude Code projects. In *Int. Workshop on Agentic Eng. (AGENT)*. ACM, 63–67. doi:10.1145/3786167.3788412

[23] S. Dueñas, V. Cosentino, J. M. Gonzalez-Barahona, A. del Castillo San Felix, D. Izquierdo-Cortazar, L. Cañas-Díaz, and A. Pérez García-Plaza. 2021. GrimoireLab: A toolset for software development analytics. *PeerJ Comput. Sci.* 7, e601 (2021). doi:10.7717/peerj-cs.601

[24] U. K. Durrani, M. Akpinar, M. Fatih Adak, A. Talha Kabakus, M. Maruf Öztürk, and M. Saleh. 2024. A decade of progress: A systematic literature review on the integration of AI in software engineering phases and activities (2013–2023). *IEEE Access* 12 (2024), 171185–171204. doi:10.1109/ACCESS.2024.3488904

[25] T. Dybå, R. Prikladnicki, K. Rönkkö, C. B. Seaman, and J. Sillito. 2011. Qualitative research in software engineering. *Empir. Softw. Eng. (EMSE)* 16, 4 (2011), 425–429. doi:10.1007/s10664-011-9163-y

[26] M. Felderer and G. H. Travassos. 2020. The evolution of empirical methods in software engineering. In *Contemporary Empirical Methods in Software Engineering*. Springer, 1–24. doi:10.1007/978-3-030-32489-6_1

[27] R. Feldt and A. Magazinius. 2010. Validity threats in empirical software engineering research–An initial survey. In *Proc. Int. Conf. Softw. Eng. & Knowledge Eng. (SEKE)*. KSI Research, 374–379. https://www.cse.chalmers.se/~feldt/publications/feldt_2010_validity_threats_in_ese_initial_survey.pdf

[28] D. Méndez Fernández and J.-H. Passoth. 2019. Empirical software engineering: From discipline to interdiscipline. *J. Syst. Softw. (JSS)* 148 (2019), 170–179. doi:10.1016/j.jss.2018.11.019

[29] S. L. France. 2024. Navigating software development in the ChatGPT and GitHub Copilot era. *Bus. Horiz.* 67, 5 (2024), 649–661. doi:10.1016/j.bushor.2024.05.009

[30] C. A. Furia, R. Torkar, and R. Feldt. 2022. Applying Bayesian analysis guidelines to empirical software engineering data: The case of programming languages and code quality. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 31, 3 (2022), 40. doi:10.1145/3490953

[31] C. A. Furia, R. Torkar, and R. Feldt. 2023. Towards causal analysis of empirical software engineering data: The impact of programming languages on coding competitions. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 33, 1 (2023), 13. doi:10.1145/3611667

[32] G. Gousios and D. Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. IEEE, 12–21. doi:10.1109/MSR.2012.6224294

[33] R. B. Grady and D. L. Caswell. 1987. *Software metrics: Establishing a company-wide program.* Prentice-Hall.

[34] L. Graf-Vlachy and S. Wagner. 2024. Cleaning up confounding: Accounting for endogeneity using instrumental variables and two-stage models. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 33, 8 (2024), 199. doi:10.1145/3674730

[35] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. 2000. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng. (TSE)* 26, 7 (2000), 653–661. doi:10.1109/32.859533

[36] H. He, R. He, H. Gu, and M. Zhou. 2021. A large-scale empirical study on Java library migrations: Prevalence, trends, and rationales. In *Proc. Europ. Softw. Eng. Conf. and the Int. Sympos. Foundations of Softw. Eng. (ESEC/FSE)*. ACM, 478–490. doi:10.1145/3468264.3468571

[37] K. Herzig and A. Zeller. 2013. The impact of tangled code changes. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. IEEE, 121–130. doi:10.1109/MSR.2013.6624018

[38] N. Hoess, C. Paradis, R. Kazman, and W. Mauerer. 2025. Does the tool matter? Exploring some causes of threats to validity in mining software repositories. In *Int. Conf. Softw. Analysis, Evolution, and Reengineering (SANER)*. IEEE, 645–656. doi:10.1109/SANER64311.2025.00067

[39] J. Howison and K. Crowston. 2004. The perils and pitfalls of mining SourceForge. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. IET, 7–11. doi:10.1049/ic:20040467

[40] S. Just, K. Herzig, J. Czerwonka, and B. Murphy. 2016. Switching to git: The good, the bad, and the ugly. In *Proc. Int. Sympos. Softw. Reliability Eng. (ISSRE)*. IEEE, 400–411. doi:10.1109/ISSRE.2016.38

[41] S. M. Hasan Kabir, Md. Tanim Rahman, and Aunik Hasan Mridul. 2025. Software defect prediction using traditional machine learning and ensemble learning algorithms. *Smart Wearable Technol.* 1 (2025), A9. doi:10.47852/bonviewswt52025645

[42] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. 2014. The promises and perils of mining GitHub. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. ACM, 92–101. doi:10.1145/2597073.2597074

[43] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empir. Softw. Eng. (EMSE)* 21, 5 (2016), 2035–2071. doi:10.1007/s10664-015-9393-5

[44] A. Khalid, G. Badshah, N. Ayub, M. Shiraz, and M. Ghouse. 2023. Software defect prediction analysis using machine learning techniques. *Sustainability* 15, 6 (2023), 5517. doi:10.3390/su15065517

[45] B. A. Kitchenham, S. L. Pfleeger, and N. Fenton. 1995. Towards a framework for software measurement validation. *IEEE Trans. Softw. Eng. (TSE)* 21, 12 (1995), 929–944. doi:10.1109/32.489070

[46] B. Kohler, D. Zollikofer, J. Einsiedler, A. Hoyle, and E. Ash. 2026. Read the paper, write the code: Agentic reproduction of social-science results. *arXiv preprint arXiv:2604.21965* (2026). doi:10.48550/arXiv.2604.21965

[47] P. Lago, P. Runeson, Q. Song, and R. Verdecchia. 2024. Threats to validity in software engineering—Hypocritical paper section or essential analysis?. In *Proc.*

Int. Sympos. Empir. Softw. Eng. and Measurement (ESEM). ACM, 314–324. doi:10.1145/3674805.3686691

[48] N. Li, M. Shepperd, and Y. Guo. 2020. A systematic review of unsupervised learning techniques for software defect prediction. *Inf. Softw. Technol. (IST)* 122 (2020), 106287. doi:10.1016/j.infsof.2020.106287

[49] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proc. IEEE* 108, 10 (2020), 1825–1848. doi:10.1109/JPROC.2020.2993293

[50] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proc. Int. Conf. Softw. Eng. (ICSE)*. ACM, 672–684. doi:10.1145/3510003.3510142

[51] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus. 2019. World of Code: An infrastructure for mining the universe of open source VCS data. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. IEEE, 143–154. doi:10.1109/MSR.2019.00031

[52] T. J. McCabe. 1976. A complexity measure. *IEEE Trans. Softw. Eng. (TSE)* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[53] A. Mockus. 2007. Large-scale code reuse in open source software. In *Int. Workshop Emerging Trends in FLOSS Research and Development (FLOSS)*. IEEE, 7. doi:10.1109/FLOSS.2007.10

[54] A. Mockus, R. T. Fielding, and J. D. Herbsleb. 2000. A case study of open source software development: The Apache server. In *Proc. Int. Conf. Softw. Eng. (ICSE)*. ACM, 263–272. doi:10.1145/337180.337209

[55] A. Mockus, R. T. Fielding, and J. D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 11, 3 (2002), 309–346. doi:10.1145/567793.567795

[56] A. Mockus and L. G. Votta. 2000. Identifying reasons for software changes using historic databases. In *Proc. Int. Conf. Softw. Maintenance (ICSM)*. IEEE, 120–130. doi:10.1109/ICSM.2000.883028

[57] A. Mockus and D. M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Tech. J.* 5, 2 (2000), 169–180. doi:10.1002/bltj.2229

[58] S. Mujahid, D. E. Costa, R. Abdalkareem, and E. Shihab. 2023. Where to go now? Finding alternatives for declining packages in the npm ecosystem. In *Proc. Int. Conf. Autom. Softw. Eng. (ASE)*. IEEE, 1628–1639. doi:10.1109/ASE56229.2023.00119

[59] J. D. Musa. 1979. Software reliability measurement. *J. Syst. Softw. (JSS)* 1 (1979), 223–241. doi:10.1016/0164-1212(79)90023-2

[60] E. Oliveira, E. Fernandes, I. Steinmacher, M. Cristo, T. Conte, and A. Garcia. 2020. Code and commit metrics of developer productivity: A study on team leaders perceptions. *Empir. Softw. Eng. (EMSE)* 25, 4 (2020), 2519–2549. doi:10.1007/s10664-020-09820-z

[61] G. P. Oliveira, A. F. C. Moura, N. A. Batista, M. A. Brandão, A. Hora, and M. M. Moro. 2023. How do developers collaborate? Investigating GitHub heterogeneous networks. *Softw. Qual. J.* 31, 1 (2023), 211–241. doi:10.1007/s11219-022-09598-x

[62] L. F.M. Osorio, P. de A. dos Santos Neto, G. Avelino, and W. A.L. Lira. 2025. An evaluation of the impact of code generation tools on software development. In *Simpósio Brasileiro de Sistemas de Informação (SBSI)*. SBC, 625–634. doi:10.5753/sbsi.2025.246605

[63] R. Pandey, P. Singh, R. Wei, and S. Shankar. 2024. Transforming software development: Evaluating the efficiency and challenges of GitHub Copilot in real-world projects. *arXiv preprint arXiv:2406.17910* (2024). doi:10.48550/arXiv.2406.17910

[64] S. L. Pfleeger and B. A. Kitchenham. 2025. Evidence-based software engineering guidelines revisited. *IEEE Trans. Softw. Eng. (TSE)* 51, 3 (2025), 814–819. doi:10.1109/TSE.2025.3526730

[65] A. Pietri, D. Spinellis, and S. Zacchiroli. 2020. The Software Heritage graph dataset: Large-scale analysis of public software development history. In *Proc. Int. Conf. Mining Softw. Repositories (MSR)*. ACM, 1–5. doi:10.1145/3379597.3387510

[66] L. H. Putnam and W. Myers. 1991. *Measures for excellence: Reliable software on time, within budget.* Prentice-Hall.

[67] P. Rani, F. Petrulio, and A. Bacchelli. 2024. On refining the SZZ algorithm with bug discussion data. *Empir. Softw. Eng. (EMSE)* 29, 5 (2024), 115. doi:10.1007/s10664-024-10511-2

[68] S. Rapaport, L. Pautet, S. Tardieu, and S. Zacchiroli. 2025. Altered histories in version control system repositories: Evidence from the trenches. In *Proc. Int. Conf. Autom. Softw. Eng. (ASE)*. IEEE, 2184–2195. doi:10.1109/ASE63991.2025.00181

[69] R. Robbes, T. Matricon, T. Degueule, A. Hora, and S. Zacchiroli. 2026. Agentic much? Adoption of coding agents on GitHub. *arXiv preprint arXiv:2601.18341* (2026). doi:10.48550/arXiv.2601.18341

[70] G. Robles, A. Capiluppi, J. M. Gonzalez-Barahona, B. Lundell, and J. Gamalielsson. 2022. Development effort estimation in free/open source software from activity in version control systems. *Empir. Softw. Eng. (EMSE)* 27, 6 (2022), 135. doi:10.1007/s10664-022-10166-x

[71] J. Ruohonen and Q. Ramadan. 2025. Tracing vulnerability propagation across open source software ecosystems. In *IFIP Int. Conf. Testing Software and Systems*. Springer, 325–332. doi:10.1007/978-3-032-05188-2_21

[72] N. F. Schneidewind. 1992. Methodology for validating software metrics. *IEEE Trans. Softw. Eng. (TSE)* 18, 5 (1992), 410–422. doi:10.1109/32.135774

[73] C. B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng. (TSE)* 25, 4 (1999), 557–572. doi:10.1109/32.799955

[74] C. B. Seaman, R. Hoda, and R. Feldt. 2025. Qualitative research methods in software engineering: Past, present, and future. *IEEE Trans. Softw. Eng. (TSE)* 51, 3 (2025), 783–788. doi:10.1109/TSE.2025.3538751

[75] H. M. Shah, Q. Z. Syed, B. Shankaranarayanan, I. Palit, A. Singh, K. Raval, K. Savaliya, and T. Sharma. 2023. Mining and fusing productivity metrics with code quality information at scale. In *Proc. Int. Conf. Softw. Maintenance and Evolution (ICSME)*. IEEE, 563–567. doi:10.1109/ICSME58846.2023.00073

[76] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng. (TSE)* 37, 6 (2011), 772–787. doi:10.1109/TSE.2010.81

[77] D. I. K. Sjøberg and G. R. Bergersen. 2023. Construct validity in software engineering. *IEEE Trans. Softw. Eng. (TSE)* 49, 3 (2023), 1374–1396. doi:10.1109/TSE.2022.3176725

[78] D. I. K. Sjøberg, T. Dybå, and M. Jørgensen. 2007. The future of empirical methods in software engineering research. In *Future of Softw. Eng. (FOSE)*. IEEE, 358–378. doi:10.1109/FOSE.2007.30

[79] J. Streit and L. Feye. 2024. Benchmarking ongoing development output in real-life software projects. In *Int. Conf. Product-Focused Softw. Process Improvement (PROFES)*. Springer, 19–34. doi:10.1007/978-3-031-78392-0_2

[80] T. Tahir, C. Gencel, G. Rasool, T. Umer, J. Rasheed, S. F. Yeo, and T. Cevik. 2023. Early software defects density prediction: Training the international software benchmarking cross projects data using supervised learning. *IEEE Access* 11 (2023), 141965–141986. doi:10.1109/ACCESS.2023.3339994

[81] V. Terragni, A. Vella, P. Roop, and K. Blincoe. 2025. The future of AI-driven software engineering. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 34, 5 (2025), 120. doi:10.1145/3715003

[82] C. Treude and M.-A. Storey. 2025. Generative AI and empirical software engineering: A paradigm shift. In *Int. Conf. AI-powered Softw. (AIware)*. IEEE, 233–239. doi:10.1109/AIware69974.2025.00033

[83] R. Verdecchia, E. Engström, P. Lago, P. Runeson, and Q. Song. 2023. Threats to validity in software engineering research: A critical reflection. *Inf. Softw. Technol. (IST)* 164 (2023), 107329. doi:10.1016/j.infsof.2023.107329

[84] C. E. Walston and C. P. Felix. 1977. A method of programming measurement and estimation. *IBM Syst. J.* 16, 1 (1977), 54–73. doi:10.1147/sj.161.0054

[85] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 1069–1088. doi:10.18653/v1/2023.emnlp-main.68

[86] M. Watanabe, H. Li, Y. Kashiwa, B. Reid, H. Iida, and A. E. Hassan. 2026. On the use of agentic coding: An empirical study of pull requests on GitHub. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* (2026). Just accepted. doi:10.1145/3798166

[87] T. Weber, M. Brandmaier, A. Schmidt, and S. Mayer. 2024. Significant productivity gains through programming with large language models. *Proc. ACM Hum.Comput. Interact. (HCI)* 8, EICS (2024), 256. doi:10.1145/3661145

[88] E. J. Weyuker. 1988. Evaluating software complexity measures. *IEEE Trans. Softw. Eng. (TSE)* 14, 9 (1988), 1357–1365. doi:10.1109/32.6178

[89] Y. Wu, Y. Wang, Y. Li, W. Tao, S. Yu, H. Yang, W. Jiang, and J. Li. 2025. An empirical study on commit message generation using LLMs via in-context learning. In *Proc. Int. Conf. Softw. Eng. (ICSE)*. IEEE, 553–565. doi:10.1109/ICSE55347.2025.00091

[90] T. Xiao, Y. Fan, F. Calefato, C. Treude, R. G. Kula, H. Hata, and S. Baltes. 2026. Self-admitted GenAI usage in open-source software. *IEEE Trans. Softw. Eng. (TSE)* 52, 6 (2026), 1891–1910. doi:10.1109/TSE.2026.3681886

[91] Y. Xu and L. Y. Yang. 2026. Scaling reproducibility: An AI-assisted workflow for large-scale replication and reanalysis. *arXiv preprint arXiv:2602.16733* (2026). doi:10.48550/arXiv.2602.16733

[92] P. Xue, L. Wu, Z. Yu, Z. Jin, Z. Yang, X. Li, Z. Yang, and Y. Tan. 2024. Automated commit message generation with large language models: An empirical study and beyond. *IEEE Trans. Softw. Eng. (TSE)* 50, 12 (2024), 3208–3224. doi:10.1109/TSE.2024.3478317

[93] A. Zerouali, T. Mens, A. Decan, and C. De Roover. 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empir. Softw. Eng. (EMSE)* 27, 5 (2022), 107. doi:10.1007/s10664-022-10154-1

[94] Y. Zhang, Z. Qiu, K.-J. Stol, W. Zhu, J. Zhu, Y. Tian, and H. Liu. 2024. Automatic commit message generation: A critical review and directions for future work. *IEEE Trans. Softw. Eng. (TSE)* 50, 4 (2024), 816–835. doi:10.1109/TSE.2024.3364675

[95] D. Zhou, Y. Wu, L. Xiao, Y. Cai, X. Peng, J. Fan, L. Huang, and H. Chen. 2019. Understanding evolutionary coupling by fine-grained co-change relationship analysis. In *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE, 271–282. doi:10.1109/ICPC.2019.00046

[96] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. 2025. A retrospective on mining version histories to guide software changes. *IEEE Trans. Softw. Eng. (TSE)* 51, 3 (2025), 842–847. doi:10.1109/TSE.2025.3533559

Received 2026-05-13; accepted 2026-07-02