Treating Run-time Execution History as a First-Class Citizen: Co-Versioning Run-time Behavior alongside Code

Marcus Kessel
marcus.kessel@uni-mannheim.de
University of Mannheim
Mannheim, Baden-Württemberg, Germany

Abstract

Behavioral Co-Versioning remains absent from mainstream practice: while developers routinely version source code with Git, they rarely persist and query how run-time behavior evolves across revisions. This paper argues that this mismatch contributes to a blind spot in software evolution analysis and CI, where rich execution information is discarded and typically reduced to pass/fail outcomes — despite partial test oracles, flakiness, and silent output or performance drift. We propose *Behavioral Co-Versioning*, a paradigm that couples the Git history with a *Behavioral Archive*: an append-only, queryable store of selected run-time observations (e.g., method I/O and performance signals) collected during test runs and keyed by commit and test context. This enables semantic diffing, behavior-aware regression localization, and retrospective auditing by querying historical executions, complementing proactive, signal-specific monitoring tools. We first outline a minimal data model and change diagnostics based on code/test/behavior fingerprints, and then demonstrate feasibility with a laptop-scale prototype that replays historical commits of a Python project, archives run-time observations in a local Parquet-backed store, and detects behavioral changes not apparent from textual diffs.

CCS Concepts

• **Software and its engineering → Software verification and validation**.

Keywords

testing, mining, oracle, evolution, behavior, analytics, repository

ACM Reference Format:
Marcus Kessel. 2026. Treating Run-time Execution History as a First-Class Citizen: Co-Versioning Run-time Behavior alongside Code. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3805589

1 Introduction

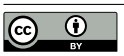
For years, approaches for Mining Software Repositories (MSR) have analyzed software evolution primarily through *static* artifacts. Mining ASTs, code diffs, and commit messages yields deep insights into developer intent and structural change, but it is inherently limited in capturing *dynamic semantics* (i.e., actual run-time behavior) due to Rice's Theorem [14, 30]. As a result, much of MSR (and, increasingly, Generative AI for SE trained on static corpora) models *what code looks like* rather than *what it does* at execution time [12, 24].

At the same time, modern Continuous Integration (CI) pipelines [16] spend substantial computing power executing test suites to generate rich run-time signals (e.g., outputs, traces, coverage, and timing). Yet standard practice discards most of these observations and reduces each run to a binary outcome: *Pass* or *Fail*. This reduction has well-known pitfalls: a passing suite can hide subtle output shifts [31], increasing flakiness [27], or silent performance regressions [15]. These issues are amplified by the oracle problem: many exercised behaviors remain unchecked due to weak or partial assertions [9, 11, 35]. Consequently, CI often validates only what developers anticipated to assert, leaving other behavioral drift (i.e., actual run-time behavior changes) invisible.

This creates a fundamental blind spot: while code versioning (e.g., Git [18]) is mature, there is no widely-adopted analogue for persisting and querying behavior over time at the granularity of code units. This is striking because the benefits of code versioning are precisely the capabilities that engineers need for behavior: stable identifiers, reviewable change artifacts, reproducibility, bisection, and rollback. However, a commit hash is an imperfect proxy for behavior: run-time outcomes can change without source edits [6] (e.g., configuration/feature flags, dependency upgrades, nondeterminism, workload drift), and even source changes can yield behavioral consequences that a partial test oracle fails to detect.

We propose *Behavioral Co-Versioning* (BeCoV): a paradigm that complements versioning code (as text) with versioning observed run-time behavior. The core idea is to couple the repository graph (e.g., Git commits) with a *Behavioral Archive* that stores behavior snapshots produced during test executions (e.g., method I/O, selected state summaries, and performance signals) keyed by revision and test context. Whereas Git tracks changes in the definition of a code unit via textual diffs, BeCoV tracks changes in its manifestation via observational data. Aligning these histories — linking code revisions to behavioral fingerprints — enables evolution analysis that is sensitive to semantics, including detecting and characterizing behavioral discrepancies even when tests pass.

Concretely, BeCoV can be understood as a *data differencing* problem — analogous to how git diff compares two versions of code: given two structured datasets of behavioral observations—one per software version—compute a meaningful, structured behavioral diff that developers can act upon. A BeCoV pipeline would (i) capture method-level inputs, outputs, and side effects during test execution; (ii) structure them at both method invocation and test granularity; and (iii) compute a behavioral diff identifying what changed,



This work is licensed under a Creative Commons Attribution 4.0 International License.
FSE Companion '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805589

was added, or removed across revisions. This dual-level design lets developers drill down: a coarse summary reveals *where* behavior shifted, while fine-grained records reveal *how*.

Existing tools in CI pipelines (e.g., unit testing, performance measurement, quality gates) are typically proactive and specific: developers must decide in advance what signals to collect (e.g., latency percentiles) and what properties to check. BeCoV is instead retrospective and generic: it preserves a reusable record of run-time observations so that new questions and new oracles can be evaluated later by querying historical runs.

We argue that BeCoV becomes feasible due to modern storage and analytics techniques (e.g., columnar formats, compression, encodings) [7, 21, 23], which suggest that structured run-time observations can be stored efficiently as a repository-integrated, queryable history rather than as semi- or unstructured ephemeral logs.

Treating CI executions as historical data enables longitudinal queries that are currently impractical, such as: *"How did the distribution of return values (or latency) of* `calculate_discount()` *change over the last 50 commits?"*. This supports semantic diffing for review/refactoring, behavioral regression localization, and forensic auditing (re-checking historical executions against newly discovered constraints without re-running old revisions).

Realizing this vision raises research challenges around (1) *volume* (capture/storage cost), (2) *identity preservation* (linking co-evolving tests to exercised code units), and (3) *representation/observability* (what to record and how to serialize it with acceptable overhead and then compare it). To ground feasibility, in this paper we present a minimal proof-of-concept on the `dateutil` Python library: we instrument a standard `pytest` suite and re-execute it across historical commits to populate a prototype behavioral archive linked to Git history, surfacing behavioral changes not apparent from static diffs alone. The remainder of this paper is structured as follows. Section 2 details the utility of BeCoV. Section 3 outlines a preliminary model and a prototype demonstration. We conclude with reflections and a call for the community to treat run-time behavior as a first-class, versioned artifact of software evolution.

2 The Case for Behavioral Co-Versioning

Existing run-time verification tools (e.g., unit testing) are valuable, but proactive and specific: developers must decide in advance which signals to instrument and which properties to assert, validating only what was anticipated at development time. BeCoV pursues a complementary goal—transforming CI executions from ephemeral checks into a queryable behavioral history—enabling retrospective analyses that are difficult to obtain from traditional CI artifacts.

We refer to the observable signals archived per code unit collectively as its *behavioral fingerprint*: input/output values, internal call sequences, side-effect summaries, and performance characteristics recorded under the test suite.

Semantic Diffing for Review and Refactoring. Textual diffs are often a poor proxy for behavioral impact, especially for refactorings that restructure code without intending to change semantics [4]. By comparing behavioral fingerprints before and after a change, BeCoV can highlight which code units exhibit observable drift and filter syntactic noise during review. Because portions of an execution trace are associated with multiple involved units, the archive also reveals cross-unit ripple effects, supporting queries such as: *"Which downstream units exhibited behavioral drift after changes to* `HelperUtil`*?"*—a behavior-centric notion of impact largely invisible to assertion-local unit testing.

Behavior-Aware Regression Localization. For two consecutive green builds, methods may return different values for the same inputs, call sequences may be reordered, and new dependencies may be introduced — all without any test failing. Such silent behavioral changes arise whenever assertions cover only a subset of observable behavior (i.e., partial oracles [9]). For example, when a developer asserts only on the final return value of a method, or on a subset of an object's state, any change to intermediate computations, internal call sequences, or side effects may remain invisible to the test verdict. A behavioral archive surfaces these shifts as fingerprint diffs even when CI remains green, and enables localization by comparing fingerprint distributions across revisions. More broadly, project-level histories can yield implicit behavioral baselines (e.g., stable output schemata or performance envelopes) that complement explicit assertions and flag anomalous drift automatically.

Retrospective Auditing and Forensics. A behavioral archive enables post-hoc evaluation of properties that were not encoded as assertions at development time. After a vulnerability report or a newly introduced compliance constraint, a team can query archived observations to assess when a problematic behavior first appeared and how broadly it manifested—without rebuilding and rerunning historical revisions, which is often impeded by dependency rot.

Downstream Opportunities. A standardized behavioral archive that continually grows (as proposed in [24] for enabling Morescient GAI) also provides potential training and evaluation data for execution-aware developer tools (e.g., AI agents suggesting missing assertions from observed invariants, or synthesizing regression tests from historical input/output patterns and historically fragile boundary cases). We view these as downstream beneficiaries of the archived information rather than its primary motivation.

3 Model and Minimal Prototype

This section sketches a concrete realization of BeCoV and provides preliminary evidence via a minimal prototype study. The key idea is to couple the Git commit history with a behavioral archive (inspired by [21]): an append-only collection of execution observations produced by CI/test runs and keyed by revision and test context. In contrast to CI artifacts that are typically ephemeral (pass/fail, raw logs), the archive treats executions as a persistent, structured dataset that supports longitudinal, behavior-aware queries. The prototype emphasizes *end-to-end feasibility* (capture → store → query) and demonstrates behavior-aware change classification. Richer query models and validation of derived labels remain open for exploration.

3.1 Conceptual Model

BeCoV aligns two complementary histories: (i) the *code history* (the Git DAG), and (ii) the *behavior history* (execution records indexed by commit, test, and exercised code units). A behavior record captures a selected set of observations from a test execution under a given revision (e.g., inputs/outputs at call boundaries, exceptions, and performance signals such as latency). Queries over these records

enable behavior-centric views of evolution (e.g., drift, instability) that are not visible from textual diffs alone.

We model the archive as a table of records of the form –

$$\langle Commit_ID,\ Test_ID,\ Unit_ID,\ Test_Hash,\ Unit_Hash,\ Obs,\ Obs_Hash,\ Context\rangle$$

where `Obs` is a (potentially partial) serialized observation payload (e.g., method I/O and latency), and `Obs_Hash` is a normalized fingerprint used for efficient comparison. `Test_ID`,`Test_Hash`, `Unit_ID`,`Unit_Hash` are the identified (test) code units and their hashes, and `Context` is the test context (e.g., environment). This design makes two assumptions explicit: (1) the archive captures observations under the test suite (per test procedure) in a specific context, not universal program semantics; and (2) determinism is not guaranteed [10] — hence both payloads and fingerprints may exhibit drift due to nondeterminism, environmental variation, or representation.

In the prototype, records are inspired based on the technical realization of the stimulus-response matrix (SRM) data structure proposed in [23]. For each captured invocation, we serialize (i) the stimulus (inputs) and (ii) the response (return value/exception and timing) into JSON — allowing for analytical queries over classic tabular representations. To enable longitudinal comparisons, we apply lightweight normalization (e.g., replacing execution-specific identifiers such as object instance IDs with stable placeholders like documented in [32]). The observation fingerprint `Obs_Hash` is then computed from this normalized representation; comparisons are performed using string equality.

3.2 Ingestion: Capturing Observations

A practical instantiation of BeCoV requires collecting observations with low friction and acceptable tracing overhead. In the prototype, we implement ingestion as a lightweight extension to `pytest` [25] (a popular unit testing framework for Python) that hooks into the test lifecycle and records a minimal observation schema: (i) call-boundary inputs/outputs for selected focal units, (ii) exceptions, and (iii) coarse-grained timing (latency) per invocation. The ingestion pipeline streams these observations into the behavioral archive together with the relevant code/test hashes.

Observability boundary. The prototype intentionally adopts a *minimal* observation tracing schema to reduce run-time overhead and data volume. Capturing deeper internal state (e.g., heap graphs) is possible in principle, but raises substantial representation and performance challenges. We treat the granularity of observation (and its efficient normalization) as a first-class research question rather than fixing it a priori. These open questions are addressed in our research roadmap in Section 4.

3.3 Storage and Querying Feasibility

Storing observations for every execution can be costly, but recent data-management techniques make persistent archival increasingly plausible. We adopt a data lakehouse layer [7] for the behavioral archive, inspired by the "observation lakehouse" style persistence layer proposed in [20, 21]: observations are serialized into tables in terms of columnar files (using the Parquet format [3]), and partitioned [2] to support selective access to code units and their run-time behavior (avoiding full table scans). Columnar compression and encoding offered by columnar storage can reduce storage overhead when tests repeatedly yield identical or highly similar observations. Queries are executed directly over Parquet using an embedded analytical engine (DuckDB [1]), avoiding a dedicated server and enabling interactive analysis in developer-local settings.

Identity Preservation. Developers typically reason about tests, but BeCoV must attribute observations to the *functional abstractions* those tests exercise (i.e., the code units that actually deliver the behavior under scrutiny). This attribution problem has two facets.

First, given a test execution, which code units constitute the *focal units* of interest versus incidental infrastructure (logging, serialization, framework glue) (cf. [17, 33])? Heuristics such as package boundaries, or naming conventions offer starting points, but no single strategy is universally reliable.

Second, once focal units are identified, their identity must be preserved across revisions (as in code versioning [26, 34]): methods are renamed, classes are split, and tests themselves co-evolve. Without robust lineage tracking, behavioral diffs risk comparing non-corresponding units and producing misleading change reports.

3.4 Behavior-aware Change Classification

Given successive revisions, we can compare code and observation fingerprints to obtain a lightweight diagnostic of how a unit and its tests co-evolve. For a fixed (`Test_ID`, `Unit_ID`), comparing revision t to $t-1$ yields the following fundamental set of behavior-aware change categories:

- **Observed behavior preserved** ($Test{=}$, $Code\Delta$, $Obs{=}$): code changed but archived observations did not change under the chosen observation schema (candidate refactoring/optimization, or behavior not captured by the schema).
- **Observed behavioral drift** ($Test{=}$, $Code\Delta$, $Obs\Delta$): code changed and observations changed (candidate regression or intended behavior change).
- **Instability / nondeterminism** ($Test{=}$, $Code{=}$, $Obs\Delta$): code did not change but observations changed (candidate flakiness, nondeterminism, environment drift, or representation noise).
- **Co-evolution** ($Test\Delta$, $Code\Delta$): both test and code changed, complicating direct drift attribution.

We emphasize that these categories are diagnostic heuristics over observed executions, not ground-truth labels or proofs of semantic equivalence — hence they may serve as additional developer feedback sent to developers as part of CI. A key research direction is to develop robust query patterns and validation methodologies that distinguish true behavioral change from observational artifacts.

Minimal Feasibility Study. To ground feasibility, we implemented a minimal pipeline for the `dateutil` Python library [29]. A lightweight `pytest` extension captures method-level inputs, outputs, and latency for heuristically identified focal units, writing observation records to a local Parquet-backed columnar store with DuckDB. A SQL-based diff engine then compares behavioral snapshots across consecutive commits. The prototype is intentionally minimal: its purpose is to demonstrate that behavioral observations *can* be captured and diffed within an existing test workflow, not to evaluate

effectiveness at scale. A thorough empirical evaluation—including quantitative characterization of detected changes, storage overhead, and developer utility—is the subject of ongoing work. For this proof-of-concept study we partition by repository and fully qualified names of focal units (e.g., `dateutil.parser.parse`) and use heuristic attribution of tests to focal units. This design prioritizes simplicity and queryability; it does not implement robust lineage tracking across complex refactorings.

Preliminary observations. Across `dateutil`'s git history (past 100 commits successfully replayed; 28.136 unique test units; 935 focal code units), we populated a behavioral archive of size ≈131MiB (the focal unit `dateutil.parser._parser.parse` accounted for ≈104MiB), and executed the behavior-aware change classification (SQL query) in 434ms on a commodity laptop. The query produced instances of the change categories. In this vision paper, we do not claim these instances are corresponding to ground-truth; they may also reflect environmental differences during replay or limitations of the observation representation. Further investigation with more controlled experimental conditions would be needed to determine whether the classifications are valid. Nevertheless, the result demonstrates the central premise of BeCoV: once execution observations are archived and keyed to commits, such hypotheses become *queryable* and can be investigated systematically rather than being lost after CI completes. For space reasons, we omit implementation details and additional query examples. The prototype, scripts, and datasets are available for inspection in the accompanying artifact [22].

4 Discussion, Related Work, and Conclusion

BeCoV sits between MSR (mining versioned static artifacts), dynamic analysis [8, 10]/regression/differential testing (collecting traces for immediate V&V, including fault localization [36]) [6, 28], and behavioral data infrastructure (e.g., lakehouse-style observation storage). Prior work provides datasets and monitoring techniques, but largely lacks a commit-aligned, queryable behavioral history for longitudinal tasks such as semantic diffing, behavior-aware regression localization, and retrospective auditing. BeCoV reframes CI executions as a persistent data asset rather than an ephemeral (quality) gate or guardrail. This shift comes with limitations and trade-offs that delimit the vision and suggest a research agenda.

Limitations and Trade-offs. **Coverage.** BeCoV is inherently based on execution: code that is not exercised (by tests or probes) has no behavioral history. In practice, many projects maintain substantial automated test suites, yet test coverage is often incomplete and unevenly distributed across code units [19]. We therefore view BeCoV as complementary to static techniques, and as increasingly viable as automated test generation and probing reduce uncovered code units. **Economics.** At first glance, BeCoV appears to conflict with an industry trend toward *test reduction* (selection/prioritization/minimization [13]) to save CI time and compute. We argue that BeCoV is orthogonal: even when the same reduced set of tests is executed, retaining selected execution observations can amortize its cost by enabling post-hoc queries (e.g., auditing, regression localization) without repeated re-execution and manual reproduction. In this sense, BeCoV shifts effort from repeated *compute-time* and *developer-time* expenditures toward a controlled, explicit *data retention* cost. **Observability.** Capturing (tracing) "more behavior" increases overhead. A practical BeCoV system must support configurable observation schemata (e.g., I/O summaries, latency, exceptions) and normalization mechanisms, while acknowledging that archived observations represent behavior *under recorded test contexts*, not universal program semantics. **Nondeterminism.** Behavioral drift may reflect nondeterminism (at value and sequence level), dependency changes, or platform variation rather than source edits. BeCoV does not eliminate these factors; instead, it makes them measurable and queryable, enabling explicit analysis of instability.

Scalability Challenges. In large-scale systems with large test sets and frequent daily commits, naive capture-everything strategies are likely infeasible. Three design dimensions shape the storage–precision trade-off: *observation depth* (how deep into the call chain to trace—shallow observation reduces volume, but may miss transitive changes—finding a balance between tracing too much vs. too little); *change-aware indexing* (e.g., structural fingerprinting to avoid revisiting unchanged portions before diffing); and *serialization profiles* (normalizing non-deterministic input/output values such as timestamps or memory addresses that otherwise introduce diff noise). On the infrastructure side, the lakehouse-style architecture described in Section 3 naturally maps onto cloud object stores (e.g., Amazon S3 storage [5, 7]); columnar Parquet files can be directly read from and written to such stores at negligible marginal cost, making long-term retention of behavioral archives economically viable even for large mono-repositories. Additionally, practical deployment requires lifecycle policies for the behavioral archive—e.g., retention windows, incremental snapshot updates, and integration with existing CI storage budgets—that remain open engineering challenges. Each dimension presents an open trade-off between precision, cost, and generality that future work must investigate.

Conclusion. With Behavioral Co-Versioning (BeCoV) and a Behavioral Archive, we have argued that run-time behavior deserves the same versioning discipline that source code receives today. Our minimal prototype suggests basic feasibility with off-the-shelf instrumentation and lakehouse-style storage, but making BeCoV as routine as source code versioning requires progress on several fronts: (1) *Identity preservation*—attributing observations to functional abstractions and tracking lineage across refactorings and test co-evolution; (2) *Representation*—designing standardized behavioral snapshot schemata, robust fingerprinting under nondeterminism (e.g., via serialization profiles), and choosing appropriate observation tracing depth; (3) *Scalable indexing*—leveraging change-aware fingerprinting structures to further improve storage and comparison; (4) *Workflow integration*—low-friction capture and querying in CI/IDE settings, with clear cost controls; and (5) *Branching and merging*—behavioral diffing techniques to identify branch-specific behavioral changes and potential conflicts at merge time, analogous to three-way textual merge in Git (i.e., comparing each branch's files with the ancestor to detect conflicting edits).

By surfacing behavioral changes that current CI pipelines miss and by opening avenues such as behavioral diffing for software evolution, BeCoV offers a complementary lens to the purely textual code view of software history that dominates current practice.

References

[1] 2026. DuckDB — An in-process SQL OLAP database management system. https://duckdb.org/. Accessed: 2026-03-31.

[2] 2026. Hive Partitioning – DuckDB Documentation. https://duckdb.org/docs/stable/data/partitioning/hive_partitioning. Accessed: 2026-03-31.

[3] 2026. Parquet File Format Documentation. https://parquet.apache.org/docs/file-format/. Accessed: 2026-03-31.

[4] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2021. On preserving the behavior in software refactoring: A systematic mapping study. *Information and Software Technology* 140 (2021), 106675. doi:10.1016/j.infsof.2021.106675

[5] Amazon Web Services. 2026. Amazon S3 data lakes for the lakehouse architecture of Amazon SageMaker. https://docs.aws.amazon.com/sagemaker-lakehouse-architecture/latest/userguide/s3-data-lakes.html. Accessed: 2026-04-01.

[6] Paul Ammann and Jeff Offutt. 2017. *Introduction to software testing*. Cambridge University Press.

[7] Michael Armbrust, Tathagata Das, Xian Zhu, Saeed Tabrizian, Reynold S. Xin, Ali Ghodsi, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Proceedings of the 2021 Conference on Innovative Data Systems Research (CIDR 2021)*. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[8] Thoms Ball. 1999. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 216–234. doi:10.1145/318774.318944

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[10] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702. doi:10.1109/TSE.2009.28

[11] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. 2019. Automatic test improvement with DSpot: a study with ten mature open-source projects. *Empirical Software Engineering* 24, 4 (2019), 2603–2635.

[12] Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. SemCoder: Training Code Language Models with Comprehensive Semantics Reasoning. In *Advances in Neural Information Processing Systems*, Vol. 37. Curran Associates, Inc., 60275–60308. https://proceedings.neurips.cc/paper_files/paper/2024/file/6efcc7fd8efeee29a050a79c843c90e0-Paper-Conference.pdf

[13] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 235–245. doi:10.1145/2635868.2635910

[14] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*. 24–27.

[15] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2010. Mining Performance Regression Testing Repositories for Automated Performance Analysis. In *2010 10th International Conference on Quality Software*. 32–41. doi:10.1109/QSIC.2010.35

[16] Martin Fowler. 2006. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html Accessed: 2026-04-01.

[17] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. 2015. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 61–70. doi:10.1109/SCAM.2015.7335402

[18] Git Development Team. 2026. Git Documentation. https://git-scm.com/docs. Accessed: 2026-03-31.

[19] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. doi:10.1145/2568225.2568271

[20] Marcus Kessel. 2025. *Observation Lakehouse: A Python library for storing and querying stimulus–response observations.* https://github.com/SoftwareObservatorium/observation-lakehouse Accessed: 2026-03-31.

[21] Marcus Kessel. 2026. Towards Observation Lakehouses: Living, Interactive Archives of Software Behavior. arXiv:2512.02795 [cs.SE] https://arxiv.org/abs/2512.02795 to appear in 2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'26).

[22] Marcus Kessel. 2026. Treating Run-time Execution History as a First-Class Citizen: Co-Versioning Run-time Behavior alongside Code. doi:10.5281/zenodo.19398211 Prototype and Dataset.

[23] Marcus Kessel and Colin Atkinson. 2024. Promoting open science in test-driven software experiments. *Journal of Systems and Software* 212 (2024), 111971. doi:10.1016/j.jss.2024.111971

[24] Marcus Kessel and Colin Atkinson. 2025. Morescient GAI for Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 123 (May 2025), 17 pages. doi:10.1145/3709354

[25] Holger Krekel and pytest-dev Team. 2025. pytest — The pytest documentation (stable). https://docs.pytest.org/en/stable/. Accessed: 2025-10-22.

[26] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2023. HyperDiff: Computing Source Code Diffs at Scale. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 288–299. doi:10.1145/3611643.3616312

[27] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. doi:10.1145/2635868.2635920

[28] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[29] Gustavo Niemeyer, Tomi Pieviläinen, Yaron de Leeuw, Paul Ganssle, et al. 2024. *dateutil: Useful extensions to the standard Python datetime features.* GitHub. https://github.com/dateutil/dateutil/ Accessed: 2026-01-22.

[30] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[31] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 90–99. doi:10.1109/ICST.2011.32

[32] Software Observatorium Documentation. 2026. SSN – Sequence Sheet Notation (Version 0.2). https://softwareobservatorium.github.io/web/docs/datastructures/ssn/. Accessed: 2026-03-31.

[33] Jeongju Sohn and Mike Papadakis. 2022. CEMENT: On the Use of Evolutionary Coupling Between Tests and Code Units. A Case Study on Fault Localization. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. 133–144. doi:10.1109/ISSRE55969.2022.00023

[34] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 908–911. doi:10.1145/3236024.3264598

[35] Masoumeh Taromirad and Per Runeson. 2025. Assertions in software testing: survey, landscape, and trends. *International Journal on Software Tools for Technology Transfer* 27, 1 (2025), 117–135.

[36] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. doi:10.1109/TSE.2016.2521368