ForkOps: Operating Fleets of Generated Software Variants

Laurent Bindschaedler✉
Max Planck Institute for Software Systems
Saarbrücken, Germany
bindsch@mpi-sws.org

Abstract

Code generation with large language models (LLMs) is making specialization cheap: file systems synthesized from specifications, database engines generated per workload, services regenerated per customer instead of configured. This paper asks the question that arrives immediately after: what are the operational problems of owning the resulting fleet of variants? A fork is no longer a copy event with version-control lineage: a variant's identity lives in its *derivation record*, the specification, kernel, and generator that produced it. The record is the genotype, the emitted code only a phenotype. Regeneration is nondeterministic expression, so two guarantees that existing tooling relies on break at once: derivation relatedness no longer implies code correspondence, and re-deriving from a fixed input preserves only what the specification pins down. Clone detection and diff-based patch porting lose their basis, and we find no infrastructure that joins identity, patching, and assurance across a population of variants. Product lines, model-driven co-evolution, and supply-chain provenance each supply pieces, but none integrates them for an independently operated fleet. This paper maps the problem space, proposes four primitives, and states a falsifiable agenda.

CCS Concepts

• **Software and its engineering → Software evolution**; **Software configuration management and version control systems**; *Maintaining software.*

Keywords

software variants, forking, LLM code generation, software genealogy, patch propagation, maintenance

ACM Reference Format:
Laurent Bindschaedler. 2026. ForkOps: Operating Fleets of Generated Software Variants. In *Proceedings of the 2nd International Workshop on Software Genomics (SWGeno '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3844178.3844641

1 Introduction

A growing body of work generates software from specifications and treats the implementation as disposable: file systems synthesized from specs [10] and database engines generated per workload [23]. Recent work argues that the specification, not the code, is the durable artifact [12], and industrial toolchains already treat it as the artifact developers edit first [3].

The economics are checkable against published evidence: generation makes implementations cheap [24], while maintenance stays dominated by comprehension: developers spend over half their time understanding code [25], and generated code is no easier to understand or trust [13]. When regenerating from a specification costs less than *rehydration*, the effort of rebuilding enough understanding of an existing implementation to change it safely, measurable as the time to a first safe change by an engineer who did not write it, specialization beats configuration. How fast organizations reach the crossover is open (RQ2, Section 5), but the direction points toward a *fleet* of variants, one per customer, workload, or device, rather than one codebase with a thousand flags.

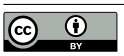
Generation research, and the verification [26] and variability [4, 21] work alongside it, invests in the upstream questions. None of them reaches, as far as we know, the one the owner of two hundred variants faces on day one: *what are these things, how do they relate, how do they evolve, and who patches them all by Friday?*

These questions had answers when forks were copies: version control recorded lineage, relatedness meant shared history, a patch ported via `diff3` because fork and mainline were textually similar, and the person who forked understood it. Two decades of research on forking and clone-and-own reuse [1, 8], missed patches [18], and automated patch porting [16, 27] rest on those assumptions. Generation weakens every one: variants are derived rather than copied, no version-control edge connects them, and often no human ever held the variant in their head.

In this workshop's terms, the units of software heredity have changed. The operational problems this creates are not new in name: product lines evolve, provenance systems track derivation, model registries version derivatives. What is new is that probabilistic regeneration breaks the two guarantees they all rely on: *derivation relatedness no longer implies code correspondence*, and *re-deriving from a fixed input preserves only what the specification pins down.* The second is deliberately weak because behavior is recoverable where the specification forces it, and pinning generator versions narrows the variance further. Neither closes the gap itself, which is a property of specification coverage rather than of sampling. We call the discipline that must operate on a population of variants under those broken guarantees ForkOps. We contribute a map of its problem space (Section 3), four candidate infrastructure primitives (Section 4), and a falsifiable research agenda (Section 5).

2 Why Fleets, Not Platforms

The *forkability trilemma*, shown in Figure 1, explains why cheap generation pushes organizations toward fleets rather than platforms. An organization wants three properties: *one shared upstream* (a single artifact whose fix reaches every deployment), *high variability*



This work is licensed under a Creative Commons Attribution 4.0 International License.
SWGeno '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3002-3/2026/10
https://doi.org/10.1145/3844178.3844641

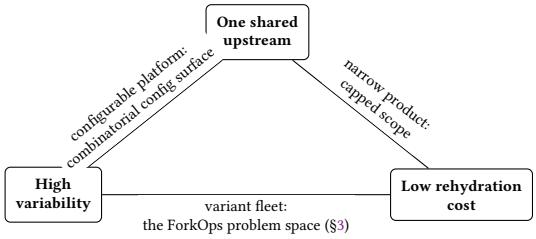


Figure 1: The forkability trilemma (conjectured): at most two corner properties hold at scale, so each edge is one achievable pair anchored in existing practice. Cheap generation pulls organizations toward the bottom edge, whose cost, fleet operations, is this paper's subject.

(per-customer, workload, or device behavior that is not a configuration choice), and *low rehydration cost* (a per-variant surface an engineer can comprehend before changing it). We conjecture that at scale it can hold at most two, and each edge is an existing practice rather than a hypothetical. Upstream plus variability is the configurable platform, where behavior lives in the cross-product of code and configuration and comprehension becomes a permanent tax [8]. Upstream plus low rehydration is the narrow product, which holds both by refusing variability. Variability plus low rehydration is clone-and-own, the corner the field already has data on: forked families diverge, duplicate effort, and miss each other's patches [1, 20]. What kept those fleets small was the cost of producing and re-understanding each member. Cheap generation removes that floor, and its signature is already measurable: duplicated code has roughly quadrupled in large repositories as assistants spread [2]. The cost changes shape rather than disappearing: from a configuration surface into *fleet operations*.

3 The ForkOps Problem Space

A *fleet*, for us, is not any batch of generated outputs but variants with independent operational lifecycles: deployed, patched, verified, or retired on separate schedules. This need does not arise for reproducible outputs redeployed atomically from one source. The problems below arise once that independence sets in, and we organize them along the fleet lifecycle: identity (P1), inventory (P2), evolution (P3), patching (P4), assurance (P5).

3.1 P1: What Is a Fork? Identity and Relatedness

Operational tooling for software families reconstructs relatedness from repository lineage or artifact similarity, and generation undercuts both. There is no copy event: a variant is produced by a derivation, roughly $variant = \mathcal{G}(spec, kernel, generator)$, where *kernel* stands for any shared upstream: a code kernel, a spec template, a policy library. This relationship lives in the overlap of the derivation inputs, not in a version-control graph.

The genomic analogy is useful but deliberately partial. The derivation record is the variant's *genotype*, the emitted code a *phenotype*, and regeneration nondeterministic expression: the same model given the same task returns syntactically and even semantically different code across runs [15]. Two expressions of the same genotype should thus differ textually while variants of unrelated intent share generated boilerplate, so phenotype similarity fails in both directions as a family signal, reporting siblings as strangers and strangers as siblings. If so, clone detection, repository mining, and provenance tools calibrated on human-authored families will misclassify generated ones. The test for this is RQ1 (Section 5).

We therefore need relatedness defined over derivation records: variant B is a fork of A when their derivations share a spec ancestor, or share a kernel beyond what two *unrelated* derivations from the same generator already share, a baseline RQ1 can measure rather than assume. Divergence then becomes a vector ($\Delta spec$, $\Delta kernel$, $\Delta generator$, $\Delta code$) rather than a diff, leaving open which components predict the quantities operators care about: whether a patch transfers, how far behavior drifted, what a rebuild costs. Versioning is unsettled for the same reason: a variant version must bind a position in derivation space to its many generated expressions rather than collapse to one artifact hash. Without P1, every later problem is ill-posed: one cannot inventory, evolve, or patch a family one cannot define.

3.2 P2: Fleet Inventory and Orphans

When generating a service takes an afternoon, variants appear without ceremony: per-team tools, per-customer glue, agent-generated "temporary" components that production comes to depend on. Forking used to be clearly visible, because a repository appeared. It is now ambient, so no registry is forced. The dangerous end state is the *orphan*: a variant whose derivation record is incomplete or stale, so it can no longer be rebuilt, only patched in place. This is the vibe-coding failure mode already visible today [21]: each in-place agent fix drifts the artifact further from any spec. A disposable fork can tolerate that drift. A fleet under customer guarantees cannot. Operators need inventory that is discovered, not declared, with rebuildability continuously checked.

3.3 P3: How Do Forks Evolve?

Evolution has three motions, and the mechanics that exist for each assume deterministic, traceable derivation.

Downstream propagation. The shared upstream (kernel, spec template, policy library) changes. Which descendants must follow? In git-based families this is a merge. Here it is a re-derivation decision per variant, and we know of no tool that computes which variants an upstream change affects, or lets a variant adopt part of one (the intent-level analog of cherry-picking).

Drift. Variants and their specs rot independently. Code-level drift is familiar. The new failure mode is *intent drift*: the business rule, legal interpretation, or environment changes while the spec and its frozen tests (the variant's *gates*) still pass, so the variant rebuilds "successfully" into the wrong system. Runtime verification and specification mining detect disagreement with an *encoded* requirement. Intent drift is the harder case: the governing requirement changed, but no spec or gate did, so nothing flags it.

Convergence. Two variants grow toward the same functionality. In the old world one merged branches. Here a code merge is meaningless (the texts are unrelated), so merging means *spec merging*: reconciling two intent histories and regenerating. The reverse matters too: deciding when n variants should collapse back into one

needs a cost model, one accounting for verification and rollout, that we have not seen proposed.

3.4 P4: How Do You Patch Forks?

Patching is the problem with a deadline, because its instances are security embargoes. A fix can originate at one of three levels: a defect in one variant, in the shared kernel (inherited by every descendant), or in the *generator* (silently injected into everything it produced). The last is the generative analog of a fleet-wide product-line defect [9], with the generator now an opaque, versioned, externally controlled dependency.

The remedy can be *ported*, *transformed*, *regenerated*, or *deferred*. Even semantic and refactoring-aware transplantation [16, 27] leans on structural correspondence between family members, which independent regeneration erodes. *Regenerating* from patched upstream sidesteps correspondence, but preserves only what the specification and its gates pin down: the rebuilt variant may differ from its predecessor in any respect the specification leaves open [6, 14], so absent trusted impact analysis, a regenerative patch is an unrequested upgrade forcing broader re-verification than the fix alone. Choosing a mechanism per variant, under an embargo clock and a bounded verification budget, is a fleet-scale optimization pairwise patch-porting cannot express, making it the sharpest problem ForkOps owns. The metric that falls out, *fleet patch latency* (time to patch, re-verify, and roll out $k\%$ of the fleet), is to ForkOps what deployment frequency is to DevOps. PaReco found missed patches endemic among a few hundred human-maintained variants [18]. Nothing suggests fleets will do better by accident.

3.5 P5: Fleet Assurance

Someone must answer for the fleet: which kernel version each variant runs on and which generator produced it, which were re-verified since the last upstream change, and which meet their obligations. For a single artifact these are maturing provenance questions (software bills of materials (SBOMs), signed builds). Fleet patch managers answer package-version compliance, but not derivation-aware exposure. *Which deployed variants derive from a kernel version older than the fixed one and have not been re-verified since the disclosure?* is the case we offer anyone who suspects ForkOps reduces to existing tools: provenance holds the derivation, product-line tooling the family, patch management the deployment, and none we know of holds the join. Assurance depends on all the others: an auditor's query joins identity (P1), inventory (P2), evolution history (P3), and patch state (P4).

4 Candidate Primitives

We do not have the solutions, only the shape they should take: artifacts and protocols, like SBOMs, that competing implementations can interoperate over.

Derivation manifest. A versioned, signed record per variant of the full derivation input: the specification, the kernel, and the generator, itself a tuple of model version, decoding parameters, agent scaffold, and tool and context set, plus the verification suite and provenance. It makes P1's derivation record concrete: the genotype as a first-class artifact. Relatedness is computed over manifests (P1), a variant without a live manifest is an orphan by definition (P2), it separates a logical variant from its many generated expressions, and it is the join key for cross-variant patching (P4) and assurance (P5). Provenance is the precedent, not the solution: in-toto [22] records how one artifact was built, and model registries version stochastic derivatives [5]. The manifest is a generated-software profile over them.

Patch lift. The fleet patch protocol (P4), stated as an interface rather than a workflow. Input: a fix, the level it originates at (variant, kernel, or generator), and the fleet's manifests. Output: per variant, a mechanism (port, transform, regenerate, or defer) and its incurred obligations. The decision rule is the open part, and RQ3 is how competing rules get compared.

Re-verification obligation set. The machine-generated list of properties (tests, proofs, policy checks, observations) that must be re-established for a given variant after a given upstream change: patch lift's unit of work, which makes fleet patching schedulable and its cost measurable.

Fleet attestation ledger. The queryable state of P5: which variants are rebuilt, re-verified, and deployed against which upstream and verifier versions. Continuous integration and delivery became a discipline when builds and deployments became inspectable objects. ForkOps becomes one when rebuild-and-prove does.

5 Research Agenda

RQ1 (P1): Does a derivation-based fork definition predict anything? Build corpora of generated variant families, constructible synthetically today, and test which divergence components ($\Delta spec$, $\Delta kernel$, $\Delta generator$, $\Delta code$) predict patch transferability and behavioral distance. Derivation-based relatedness is worth testing first because it is available before anything is run, as an embargo clock requires. If code-level or behavioral metrics predict these equally well, P1 dissolves.

RQ2 (P2): When do generated artifacts acquire independent lifecycles? Instrument generation pipelines or study early adopters longitudinally: how many generated artifacts persist, are deployed independently, receive post-generation edits, or diverge in generator or spec version? A null result (outputs stay ephemeral and are regenerated atomically) collapses ForkOps into existing build and product-line tooling. A positive result calibrates its timeline.

RQ3 (P4): Benchmark fleet patching. A seeded kernel, N variants (one-spec populations are already constructible [19]), a disclosed vulnerability, a verification harness, a budget. Compare port-all, regenerate-all, and derivation-aware per-variant mechanism selection, scored by fleet patch latency, residual exposure, and behavioral churn. This lifts pairwise patch-porting benchmarks [27] to fleets.

RQ4 (P3): Detect intent drift. Given a spec, its gates, and production traces, flag divergence between declared, enforced, and observed behavior before a rebuild bakes the drift in. Predictive failure models for coding agents [11] are a starting point. Because the governing intent can change with no artifact to mine, the hard part is supplying an external oracle for current intent (regulatory feeds, incident reports, changed dependencies).

RQ5 (P5): Fleet assurance queries. Define the auditor's query language over manifests and ledgers, and show it answerable from infrastructure alone on a synthetic fleet.

6 Related Work

ForkOps rests on established disciplines and departs from each at one assumption. Fork and clone-and-own studies documented divergence, duplicated effort, and missed patches [1, 8, 18]. Synchronization and porting tools transfer changes between variants [16, 27]. These reconstruct correspondence from copied artifacts or recorded edits, which regenerated siblings lack. Product-line engineering plans variability and analyzes family-wide change [9, 17], and recent work recasts regeneration as the derivation mechanism [4, 21]. ForkOps is the residual problem when the family is emergent, has no explicit feature model, and regenerates nondeterministically. Model-driven engineering co-evolves models and code through traceable, deterministic transformations [7]: regeneration keeps the round trip but removes the trace and the determinism it relies on. Model registries version stochastic derivatives [5] and N-version work studies populations generated from one specification [19]; both preserve correspondence our variants lack. Finally, regeneration systems demonstrate feasibility [10, 23], position work argues the spec is the durable artifact [12], and verification trusts a *single* generated artifact [26]. ForkOps takes all of these as inputs and asks the cross-variant operational questions that none, to our knowledge, poses.

7 Conclusion

The generation problem is being solved. The cross-variant operations problem is barely posed, and if specialization spreads as the economics suggest, the fleets will follow. Clone-and-own research spent two decades documenting what happens when humans fork carefully and occasionally. ForkOps is what the field must build now that machines fork carelessly and constantly. Every problem claimed here comes with a test (Section 5). We invite the community to start before the first fleet-wide incident does it for us.

Acknowledgments

The author thanks the SWGeno '26 reviewers for their comments. The author used AI tools for editorial assistance (drafting, copy-editing, and trimming for length); all technical content, results, and conclusions are the author's own.

References

[1] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. 2022. Reuse and Maintenance Practices Among Divergent Forks in Three Software Ecosystems. *Empirical Software Engineering* 27, 2 (2022), 54. doi:10.1007/s10664-021-10078-2

[2] GitClear. 2025. AI Copilot Code Quality: 2025 Data Suggests 4x Growth in Code Clones. https://www.gitclear.com/ai_assistant_code_quality_2025_research Industry report.

[3] GitHub. 2025. Spec Kit Templates. https://github.com/github/spec-kit/releases/tag/v0.0.1 Software, version 0.0.1.

[4] Sandra Greiner, Klaus Schmid, Thorsten Berger, Sebastian Krieter, and Kristof Meixner. 2024. Generative AI And Software Variability — A Research Vision. In *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '24)*. 71–76. doi:10.1145/3634713.3634722

[5] Wei Hao, Daniel Mendoza, Rafael Mendes, Deepak Narayanan, Amar Phanishayee, Asaf Cidon, and Junfeng Yang. 2024. MGit: A Model Versioning and Management System. In *Proceedings of the 41st International Conference on Machine Learning (ICML '24) (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 17597–17615.

[6] Jinhao Hu, Ashvin Goel, and Laurent Bindschaedler. 2026. Don't Trust the Code, Check Its Effects: Runtime Refinement for Regenerated Systems Code Under an Adversarial Generator. In *Proceedings of the 5th Workshop on Practical Adoption Challenges of ML for Systems (PACMI '26)*.

[7] Zohra Kaouter Kebaili, Djamel Eddine Khelladi, Mathieu Acher, and Olivier Barais. 2025. Automated Co-Evolution of Metamodels and Code. *IEEE Transactions on Software Engineering* 51, 4 (2025), 1067–1085. doi:10.1109/TSE.2025.3540545

[8] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. 432–444. doi:10.1145/3368089.3409684

[9] Martin Kuhlemann and Martin Sturm. 2010. Patching Product Line Programs. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD '10)*. 33–40. doi:10.1145/1868688.1868694

[10] Qingyuan Liu, Mo Zou, Hengbin Zhang, Dong Du, Yubin Xia, and Haibo Chen. 2026. Sharpen the Spec, Cut the Code: A Case for Generative File System with SYSSPEC. In *Proceedings of the 24th USENIX Conference on File and Storage Technologies (FAST '26)*. 291–311.

[11] Bardia Mohammadi, Lars Klein, Aman Chadha, Akhil Arora, and Laurent Bindschaedler. 2026. The Working Set of a Coding Agent: Coherence Debt in Repository-Scale Tasks. arXiv:2608.16630 [cs.SE]

[12] Martin Monperrus. 2026. Bootstrapping Coding Agents: The Specification Is the Program. *IEEE Software* 43, 4 (2026), 19–22. doi:10.1109/MS.2026.3687051

[13] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. 1–13. doi:10.1145/3597503.3639187

[14] Akanksha Narula, Mofasshara Binte Rafique, and Laurent Bindschaedler. 2026. The Invisible Lottery: How Subtle Cues Steer Algorithm Choice in LLM Code Generation. In *Proceedings of the 43rd International Conference on Machine Learning (ICML '26)*.

[15] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–28. doi:10.1145/3697010

[16] Shengyi Pan, You Wang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2024. Automating Zero-Shot Patch Porting for Hard Forks. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. 363–375. doi:10.1145/3650212.3652134

[17] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. doi:10.1007/3-540-28901-1

[18] Poedjadevie Kadjel Ramkisoen, John Businge, Brent van Bladel, Alexandre Decan, Serge Demeyer, Coen De Roover, and Foutse Khomh. 2022. PaReco: Patched Clones and Missed Patches among the Divergent Variants of a Software Family. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. 646–658. doi:10.1145/3540250.3549112

[19] Javier Ron, Benoit Baudry, and Martin Monperrus. 2026. N-Version Programming with Coding Agents. arXiv:2606.20158 [cs.SE]

[20] Ștefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME '15)*. 151–160. doi:10.1109/ICSM.2015.7332461

[21] Xhevahire Tërnava. 2026. Where Did the Variability Go? From Vibe Coding to Product Lines by Regeneration. In *Proceedings of the 1st International Conference on Software and Systems Reuse, Product Lines, and Configuration (VARIABILITY '26)*.

[22] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. 1393–1410.

[23] Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. 2026. Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines. *Proceedings of the VLDB Endowment* 19, 11 (2026), 3759–3771. https://www.vldb.org/pvldb/vol19/p3759-wehrstein.pdf

[24] Matt Welsh. 2023. The End of Programming. *Commun. ACM* 66, 1 (2023), 34–35. doi:10.1145/3570220

[25] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44, 10 (2018), 951–976. doi:10.1109/TSE.2017.2734091

[26] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 3454–3482. doi:10.1145/3763174

[27] Zhiqing Zhong, Jiaming Huang, and Pinjia He. 2026. BackportBench: A Multilingual Benchmark for Automated Patch Backporting. *Proceedings of the ACM on Software Engineering* 3, FSE (2026), 700–722. doi:10.1145/3797127

Received 2026-07-27; accepted 2026-08-21