From Instructions to Enforcement: Evaluating Guardrails for AI-Assisted Software Development

Andrew Koski

6 November 2026 | Working paper · Version 1.0 | Draft prepared 6 November 2026

Abstract

Written development instructions describe expected behavior, but their presence does not establish that a delivered change satisfies them. I argue that important, mechanically testable requirements should be connected to executable checks at the boundary where software is accepted or released. This paper proposes a controlled comparison of written instructions, ordinary tests, and additional policy gates for AI-assisted development. It separates check availability from enforcement reachability and examines bypasses, stale results, false positives, and evaluation leakage. The framework is motivated by a shared preflight system used across multiple application repositories, but it does not treat installation counts as evidence of defect reduction. The contribution is a proposed enforcement model and reproducible study design. No controlled efficacy results are reported.

Keywords: software agents; executable policy; quality gates; evaluation; software assurance

Explore the paper

1. The position
2. What counts as enforcement
3. Motivation from a shared system
4. Proposed controlled study
5. Test the bypass paths
6. What would change the conclusion
7. Conclusion
References

1. The position

A rule can be clearly written, repeatedly supplied to a coding agent, and still be absent from the final change. I argue that an important distinction is being lost when a project treats the existence of instructions as evidence of compliance. Instructions express intent. A check can test a property. An enforced boundary determines whether failure actually prevents acceptance.

Repository-level benchmarks such as SWE-bench evaluate whether systems can resolve real software issues. [1] That is a useful capability question. The question here is narrower: given an agent attempting a task, what changes when selected project requirements are independently checked before its output is accepted? A task can be functionally resolved while still violating a release requirement, and a release requirement can be poorly designed even when it is consistently enforced.

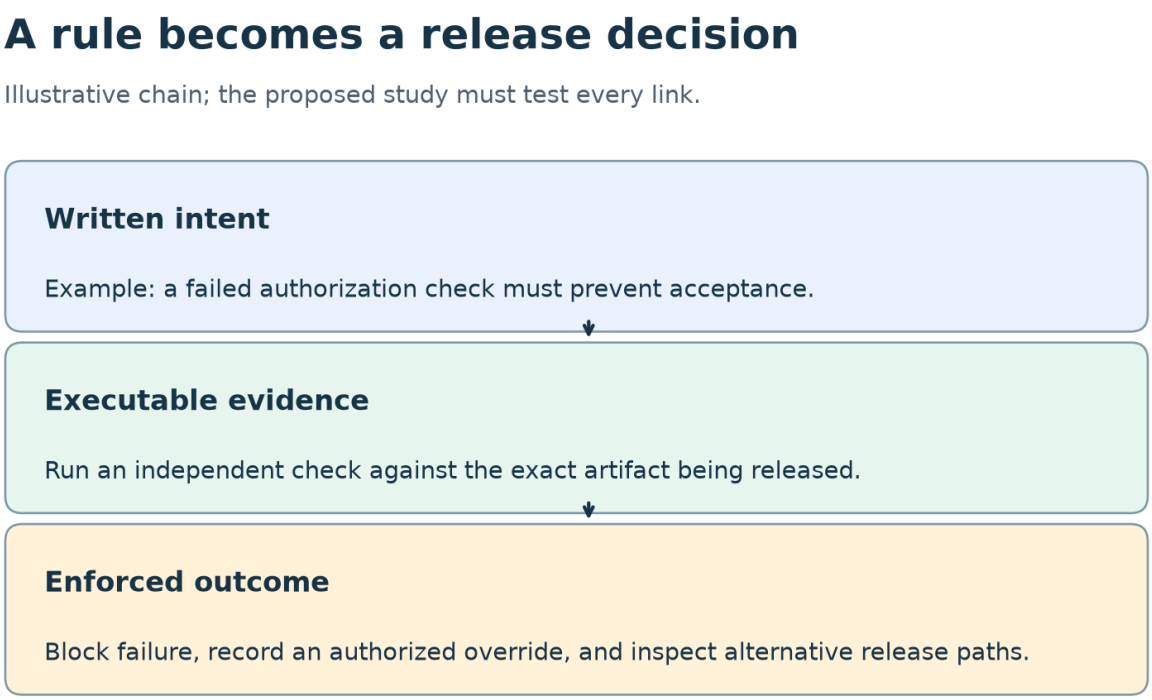


Figure 1. Illustrative path from written intent to an enforced release decision.

2. What counts as enforcement

An enforcement claim requires a named property, a check that can detect its violation, and a reachable acceptance path that honors the result. It also requires a declared scope. A local commit hook does not govern every way code can reach production. A passing check against an earlier revision does not establish the state of a later artifact.

For a specified release path, let E mean that the relevant check executes on the candidate artifact, D that it detects a violation within its declared scope, and B that a detected failure blocks acceptance. Successful prevention requires their conjunction. This decomposition is not an independence assumption or a numerical reliability estimate. It identifies separate failure modes that an audit must inspect.

The trust boundary matters. If an agent can edit the gate, its expected answers, and the mechanism that interprets failure, the evaluation may reward removal of the constraint. Some tasks legitimately require test changes; those changes should be reviewed or evaluated against held-out acceptance criteria. An ordinary test suite and an executable guardrail can overlap. Their labels are less important than their coverage and authority.

3. Motivation from a shared system

A local audit of the Empline preflight package, version 2.3.123, records required gate installation and hook wiring across 16 consumers. The audit describes checking effective hook paths, exit behavior, required modules, configuration exclusions, and collision handling. [2] This is a documented operational snapshot, not an independent evaluation of prevented incidents or a permanent guarantee about those repositories.

The distinction between installation and enforcement is the useful lesson. A check may exist but never run. A script may log an error and return success. A configuration filter may exclude the required check. A cached result may refer to a different revision. A later action may bypass the path entirely. Each possibility should be represented as a testable condition rather than absorbed into the statement that a project has guardrails.

Machine-learning system research has long emphasized maintenance risks outside the model itself, including dependencies and system interactions. [3] This paper applies a similarly system-level perspective to agent-assisted delivery. It does not claim that executable checks or continuous integration are new inventions.

4. Proposed controlled study

Use three conditions on matched repository tasks. Condition A provides written project instructions. Condition B adds the ordinary test suite and a requirement to run it. Condition C adds enforced checks for selected policies at the acceptance boundary. Keep the model, tool access, task prompt, starting revision, and resource budget matched. Repeat tasks across multiple random seeds or independent agent runs and randomize condition order.

Include both naturally occurring defects and deliberately introduced mutations. Examples include a broken authorization branch, a swallowed failure, a required command omitted from a release script, and a mismatch between validated and released artifacts. The mutation set must include defects outside the added checks' design set. Otherwise the experiment risks measuring how well a gate recognizes examples it was constructed to recognize.

Keep an independent oracle outside the agent's writable workspace. The oracle should assess functional completion and the selected policy properties. Where judgment is required, use a documented rubric and reviewers unaware of the experimental condition. Preserve task-level outcomes, unsuccessful attempts, changed tests, check outputs, and artifact hashes. A final pass count without those records is insufficient for diagnosing how a result was achieved.

Predefine the primary outcome as policy-violating artifacts accepted per completed task, alongside overall task completion. Secondary outcomes include false blocks, repair cycles, human review time, elapsed time, and compute cost. Report paired differences with uncertainty intervals, repository-level variation, and incomplete tasks. A gate that stops every change has perfect blocking but poor utility; completion and cost must remain visible.

5. Test the bypass paths

Figure 1 places evaluation at the acceptance boundary rather than solely inside an agent's conversation. The study should separately test direct pushes, alternative release commands, changed check configuration, stale results, and a legitimate emergency override. Some environments will not permit all these paths; report which are reachable instead of inventing attack capability.

An override is a policy choice, not proof that a check never mattered. It should identify its authority, reason, affected artifact, and subsequent review. The analysis should report both outcomes with overrides excluded and outcomes with the actual override rate included. Excluding every bypass from the headline result would conceal the operational behavior the system is meant to improve.

Also test clean changes. A gate that blocks a valid dependency update or incorrectly interprets an optional feature may create pressure to bypass it. False-positive rates should be measured by policy family and corrected at the owning rule rather than suppressed in the report.

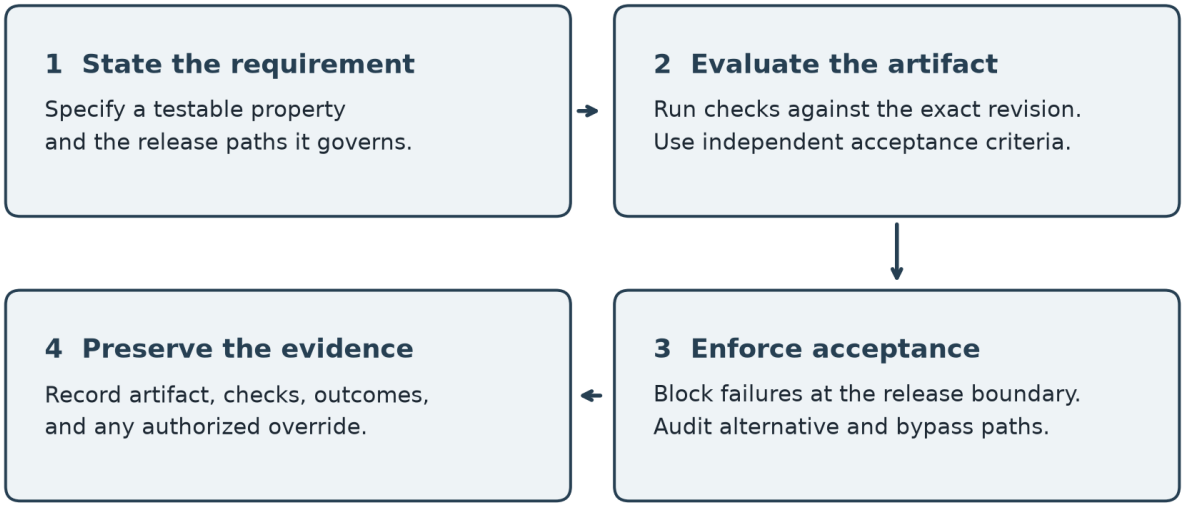


Figure 2. Proposed enforcement workflow. Passing a check is consequential only when the acceptance boundary honors it.

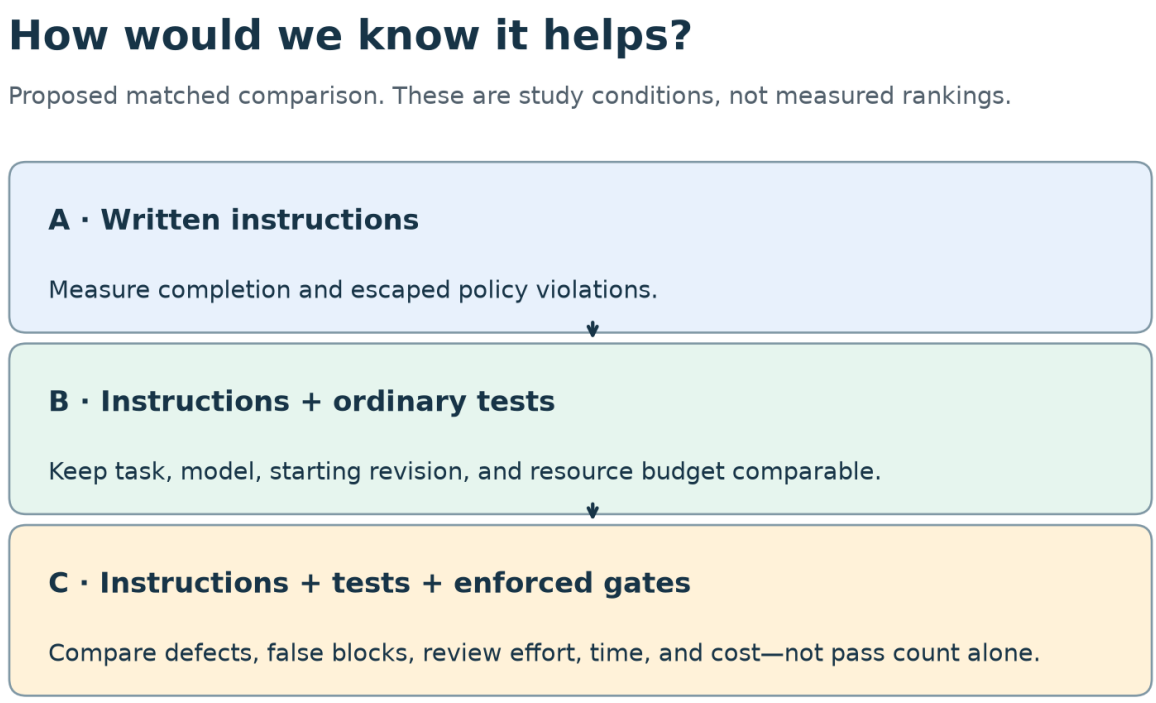


Figure 3. Proposed study conditions. No measured ranking is implied.

6. What would change the conclusion

I would reduce confidence in the value of additional gates if they fail to lower escaped policy violations at comparable task completion and review effort. If ordinary tests achieve the same result with less maintenance, they are the better intervention for that setting. If measured gains disappear on held-out repositories, the paper should describe a local engineering result rather than a general agent-safety improvement.

Threats to validity include task leakage, synthetic defects unlike real failures, inconsistent tool budgets, model-version changes, and correlated runs on similar repositories. A study of one agent and one stack cannot establish universal effectiveness. The proposed work should publish its task construction method and failure taxonomy so that another team can evaluate the same claim in a different environment.

7. Conclusion

The practical objective is to make important requirements observable at the point where they matter. Written instructions remain useful for explaining intent and guiding implementation. Executable checks can contribute independent evidence, and enforced boundaries can make that evidence consequential. The claim worth testing is whether this combination reduces escaped defects without making ordinary development unworkable. This paper defines that test; it does not presume the outcome.

References

[1] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? International Conference on Learning Representations. https://arxiv.org/abs/2310.06770

[2] Empline preflight project (2026). Required consumer gate audit. Internal engineering record, docs/CONSUMER_GATE_COVERAGE.md, snapshot dated 6 November 2026, package 2.3.123. Not independently reviewed or publicly archived; cited as operational motivation only.

[3] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015). Hidden Technical Debt in Machine Learning Systems. Advances in Neural Information Processing Systems 28. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html