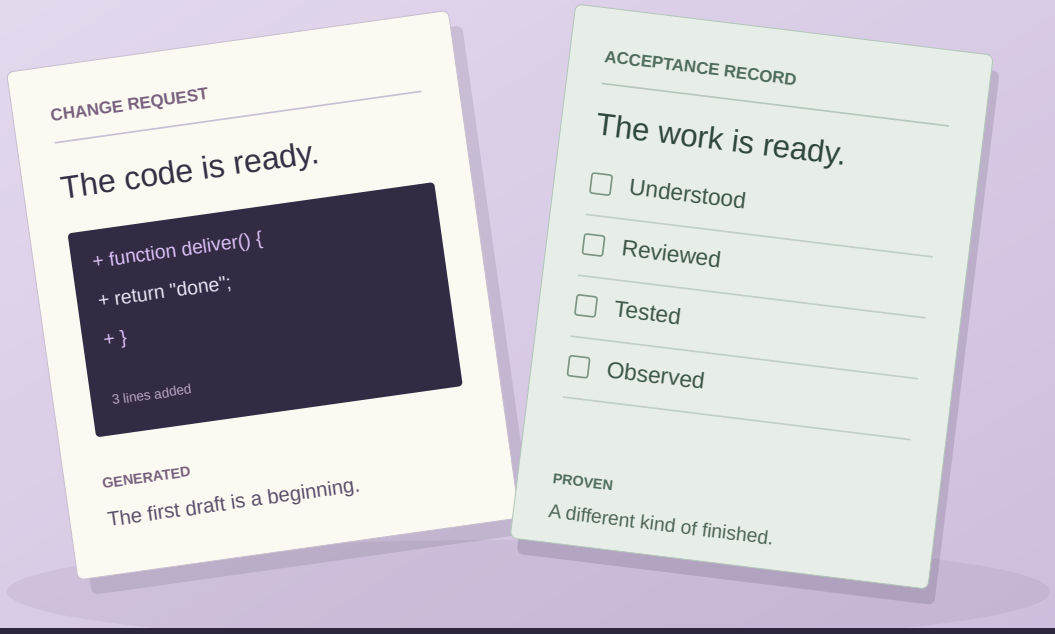
kymata labs

RESEARCH PAPER 01 / AI & SOFTWARE DELIVERY

The Code Is Cheap. The Proof Is Expensive.

AI can write the first draft. Who pays for the work of making it trustworthy?



Kymata Labs Research
2026-11-15 · Edition 1.0

Independent analysis
kymatalabs.com

Small studio. Open worlds.

THE LONGER VIEW

Executive summary.

AI coding productivity is often discussed in terms of what a tool can produce. The more useful question is what a team can accept and sustain. This paper reproduces a bounded early-2025 study result, examines what engineering reviews and coding evaluations actually measure, and develops a practical acceptance record. It separates human effort from elapsed delivery time, keeps unsuccessful attempts visible, and tests the accounting with an explicitly synthetic example. The argument leaves room for substantial gains: assistance can improve testing and review as well as drafting. Its contribution is a way to determine whether those gains survive a consistent completion standard, rather than assuming either that faster generation means cheaper delivery or that verification must consume every saving.

> Analysis of inspected primary research data, evaluation artifacts and engineering documentation. METR's early-2025 result was reproduced from its official dataset and core regression specification; it does not measure current tools or lifetime maintenance. The acceptance framework and effort scenarios are original proposals, not new experimental findings. Source access: November 16, 2026 UTC.

01 / The clock that stops too soon
02 / What an inconvenient study actually measured
03 / A patch is a proposal to the whole system
04 / The strongest case for going faster
05 / Make acceptance observable
06 / Three changes, two different economics
07 / Measure the workflow you intend to buy
08 / What this paper can and cannot establish
09 / Keep the proof attached to the change
Q&A / Questions worth asking
Sources / References

© 2026 Kymata Labs LLC. All rights reserved.
Read the web edition: kymatalabs.com/research/the-code-is-cheap-the-proof-is-expensive

The clock that stops too soon

A change can look finished at the exact moment its most consequential questions become visible. The function exists. The diff is tidy. The demonstration works. Someone still has to establish whether it solves the requested problem, respects the system around it, and can be understood when the next person changes it.

That distance between a plausible patch and a change worth accepting is where the argument about AI coding productivity belongs. A tool can produce code quickly and still leave a team with more work. It can also produce code quickly, supply useful tests, explain an unfamiliar dependency, and leave the team with less work. The visible speed of generation cannot distinguish these outcomes.

The title is an economic proposition, not a universal price comparison. As generating a candidate implementation becomes easier, the scarce resource may shift toward establishing which candidates deserve to enter a working system. “Proof” here means proportionate evidence: tests, inspection, integration checks, and a responsible acceptance decision. It does not mean mathematical proof, perfect software, or a new bureaucracy attached to every edit.

This paper argues for a change in the unit of account. Count accepted changes against explicit requirements, and keep the costs of reaching acceptance attached to them. Then follow their consequences for a declared observation period. Lines produced, suggestions accepted, and time to a first patch can help explain a workflow. They cannot, on their own, establish its productivity.

The distinction matters because different clocks can tell different truths. An author may save an hour while another person spends it reviewing. A team may reduce human effort yet wait longer for a deployment window. A release may arrive sooner while creating additional repair work afterward. These are different outcomes, and a useful evaluation keeps them separate.

The evidence assembled here does not establish that current AI tools generally slow programmers down. It establishes a narrower historical result, describes what several primary engineering artifacts actually measure, and develops an operator method for testing whether apparent savings survive acceptance. The proposed method should be capable of recognizing a substantial gain. Otherwise it would be an argument disguised as an evaluation.

What an inconvenient study actually measured

METR’s repository for *Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity* contains anonymized data for 246 issues completed by 16 developers. Of those issues, 136 allowed AI and 110 did not. The repository links its announcement to 10 July 2025. This is a historical study of experienced open-source developers and early-2025 tools, not a November 2026 verdict on coding assistants. [1]

For this paper, the published core regression was independently reproduced from the captured data using the specification in the repository. Its adjusted time effect was **18.84% longer implementation time when AI was allowed**, with a reproduced normal-approximation 95% confidence interval of **1.26%**

to 39.48% longer under the homoskedastic specification. Those numbers describe the fitted effect in this study; they are not a forecast for a reader's organization. [1]

The outcome deserves as much attention as the headline. The analysis adds initial implementation time to post-review implementation time. The README describes the former as time to put a pull request up for review and the latter as time spent fixing that pull request before merge. This clock continues past the first submission. It captures some of the work that a draft-only measurement would miss. [1]

It still does not count everything this paper recommends counting. Developer repair time is not the same as every reviewer's labor. Time spent implementing is not the same as calendar time waiting in a queue. The dataset does not establish a lifetime maintenance cost, a security defect rate, or a legal exposure. A stronger measurement boundary should not be retroactively attributed to the study simply because it supports the argument.

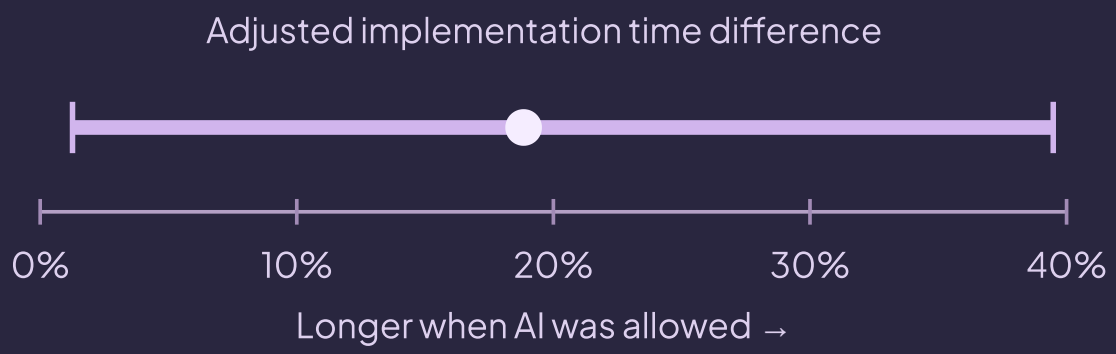
There is also a small but important reproduction detail. Sixteen post-review values are missing in the captured CSV. The published analysis fills missing values with the mean for the same treatment group, then regresses log total implementation time on AI treatment and log predicted time without AI. We used that specification rather than substituting raw group averages. A code comment's approximate missingness description does not replace the count in the inspected file. [1]

The study's value here is corrective. It shows that access to a coding tool and lower implementation time need not travel together in a real development setting. It does not identify one universal cause of that divergence. Reading the result as proof that reviewing generated code is always slower would go beyond these data; so would treating a later improvement in tools as evidence that the observed result never mattered.

FIGURE 01 HISTORICAL STUDY · EARLY 2025

A result with boundaries.

The reproduced adjusted difference in implementation time when AI was allowed, in one study of experienced open-source developers.



01

+18.84%

Adjusted time difference

Longer implementation time when AI was allowed; not an estimate for today's tools.

02

+1.26–39.48%

95% confidence interval

Normal approximation, homoskedastic specification. Sixteen developers; 246 issues.

Includes initial and post-review implementation. Does not measure all reviewer labor, waiting time or lifetime maintenance.

METR official dataset and regression; independently reproduced by Kymata Labs. [1]

A patch is a proposal to the whole system

Code review already has a vocabulary for work that generation speed does not capture. Google's published review guidance asks whether a change belongs in the codebase, whether it integrates with the rest of the system, whether it is more complex than necessary, and whether its tests are useful. Its guidance explicitly asks whether tests would fail when the code is broken. These are engineering criteria, not findings about AI. [2][3]

They help explain why a locally convincing patch can remain expensive. Consider an explicitly hypothetical change to a billing service. A generated function may correctly calculate a total for the examples in a prompt. Acceptance may also depend on rounding conventions, currency representation, retries, and the expectations of existing callers. The extra work is not an objection to generating the function. It is part of determining what function was needed.

The same distinction applies to generated tests. If an implementation and its tests encode the same mistaken interpretation of a requirement, running those tests can confirm agreement without resolving the interpretation. That is a logical failure mode, not a measured claim that machine-written tests are generally worse. An independent acceptance case must come from somewhere: a product requirement, a known regression, an external contract, or a reviewer who understands the behavior being protected.

There are therefore at least three routes by which faster generation can fail to produce faster delivery. The candidate may require correction. The candidate may be correct but too broad or unfamiliar to review economically. Or the surrounding acceptance process may become the limiting step as more candidates arrive. None requires assuming that every generated line is defective.

The third route is especially easy to overlook. In a hypothetical team with a fixed number of reviewers, doubling the arrival of changes does not double the capacity to inspect them. Some additional changes may be valuable; some may be optional work that was previously too costly to propose. In this hypothetical, a growing queue would be consistent with arrivals exceeding acceptance capacity. In an observed team, larger changes or reduced reviewer availability could also explain queue growth. The queue alone would not tell us that the new work was worthless.

The response is not to demand exhaustive inspection of every conceivable risk. Google's review standard explicitly favors changes that improve overall code health even when they are imperfect, balancing progress against quality. [2] A proportionate acceptance standard is essential to the argument. If a team quietly raises the bar only for AI-assisted work, it can manufacture an apparent productivity loss. If it lowers the bar only for that work, it can manufacture a gain.

The strongest case for going faster

The strongest objection to this thesis is practical: assistance can reduce the cost of both producing and checking a change. GitHub's Copilot Chat documentation describes generating unit tests, suggesting edge cases, proposing fixes, explaining code, and summarizing pull requests. These are vendor-documented capabilities, not independent measurements of savings. The same documentation says generated tests may miss scenarios and still require manual testing and review. [7] If the contributions survive ordinary acceptance and reduce total effort, the savings are real. An emphasis on verification must leave room for that outcome.

It would also be wrong to say that every coding evaluation stops at a generated draft. SWE-bench describes a task in which a model receives a codebase and an issue and generates a patch to resolve it; the project supplies an executable evaluation workflow using Docker. Its unit of evaluation is more substantive than text completion. The inspected repository does not establish how a particular benchmark score translates into a particular team's delivery speed. [4]

GitHub Next's public timed Copilot experiment provides another useful correction. Its instructions stop the timer when GitHub Classroom records a full passing test suite. The accompanying task asks for a toy HTTP parser and response engine, without actual network requests. It therefore includes an executable acceptance boundary. It is not merely a stopwatch on typing. The inspected repository does not provide an effect estimate, so none is claimed here. [6]

The question is how much of a reader's actual work resembles such a bounded task. When the requirement is clear, the relevant context fits, and a trustworthy check is inexpensive, faster generation has a plausible path to lower total effort. Where acceptance depends on understanding many hidden conventions, the path is harder. These are proposed conditions for local testing, not a ranking of occupations or a claim that one experiment settles the matter.

AI may also change which work gets done. A small improvement that previously failed a cost-benefit test might become worthwhile. Its benefit would disappear from an evaluation that counted only time saved on an unchanged backlog. Conversely, expanding a backlog merely because producing patches is easier can absorb the savings. A fair assessment needs both task efficiency and a judgment about the value of the tasks being accepted.

Make acceptance observable

The operator contribution of this paper is a compact acceptance record. It connects the intention, the evidence, the decision, and the costs of a change. It can live in a pull-request description and a small export. Its purpose is to make a productivity claim inspectable without requiring developers to narrate every keystroke.

Before implementation, write down the behavior the change must produce and one important behavior it must preserve. Record the checks that would distinguish success from a plausible mistake. Choose an owner for the acceptance decision and specify whether acceptance means merge, release, or successful deployment. These endpoints are not interchangeable.

Then record human effort separately from elapsed time. Human effort includes author work and review or repair effort by others, without counting the same person's interval twice. Elapsed time runs between declared events. A machine executing tests for ten minutes may add ten minutes of elapsed time and very little active labor. Two people reviewing simultaneously may add twice the labor without doubling the wait.

Record	Minimum useful content	What it prevents
Acceptance contract	Required behavior, protected behavior, decision owner	Calling a convincing demonstration a completed change
Evidence	Check names, results, inspected revision, known limits	Treating the existence of tests as evidence that they passed
Effort	Author, reviewer, and repair minutes; missing entries marked	Booking a saving for one person while ignoring transferred work
Elapsed time	Start, ready-for-review, acceptance, release where relevant	Confusing queue reduction with reduced labor
Disposition	Accepted, rejected, abandoned, or still open	Removing unsuccessful attempts from the denominator
Follow-up	Declared observation window and attributable repairs	Making maintenance disappear when the merge button is pressed

This is a proposed measurement design, not a validated standard. It should be kept small enough to use. A routine copy change may need a preview and a focused check. A stateful migration may need considerably more evidence. The criterion is the consequence of a mistake and the uncertainty of the change, applied consistently across assisted and unassisted work.

There is established precedent for extending measurement beyond coding activity. The Four Keys project defines delivery measures including lead time to production and change failure rate. Its repository is explicitly unmaintained, and its definitions are cited as an example of delivery-oriented instrumentation, not as the current complete DORA standard or a recommended deployment package. [5]

FIGURE 02 PROPOSED MEASUREMENT FRAMEWORK

Two clocks. One acceptance decision.

Human effort and calendar time can move differently. Keep both attached to the same, explicitly defined outcome.

01

People

Human effort

Author implementation + reviewer inspection + repair. Count person-time without double-counting.

02

Time

Elapsed delivery

Work begins → ready for review → acceptance → release, when release is the chosen endpoint.

03

Evidence

Acceptance contract

Required behavior, protected behavior, checks and a named decision owner.

Keep rejected and unfinished attempts visible. Declare a follow-up window; do not imply later maintenance has been measured.

Kymata Labs analytical framework, informed by Google review guidance and Four Keys. [2] [3] [5]

Three changes, two different economics

The following worked example is synthetic. It contains no customer measurements, model calls, or estimate of a typical team. Its purpose is to make the accounting testable.

Imagine three changes, A, B, and C. Without assistance, each takes 40 person-minutes to draft and 20 to check and repair. Total effort is 180 person-minutes. With assistance, drafting takes 10 minutes per change. Checking and repair take 25 minutes for A, 35 for B, and 100 for C.

Fictional change	Baseline draft	Baseline check/repair	Assisted draft	Assisted check/repair
A	40	20	10	25
B	40	20	10	35
C	40	20	10	100
Total person-minutes	120	60	30	160

Drafting effort falls by 75%. Total effort rises from 180 to 190 person-minutes, an increase of 5.6%. Both statements are true. The mistake would be to label the first number a measured improvement in the whole workflow.

Now change one assumption. Suppose a clearer interface contract and a focused regression fixture reduce C's assisted checking and repair to 40 minutes. The assisted total becomes 130 minutes, 27.8% below the baseline. Generation performance has not changed. The economics have changed because the cost of reaching acceptance has changed. Any effort required to create that contract or fixture must also be charged to the evaluation, or explicitly amortized across its expected uses.

The example does not show that verification will always eat the saving. It shows why verification is a productive place to invest. Better requirements, smaller changes, established fixtures, and visible dependencies may help assistance pay off. They can also improve unassisted work; a fair comparison must give the baseline the same engineering improvements rather than crediting all of them to AI.

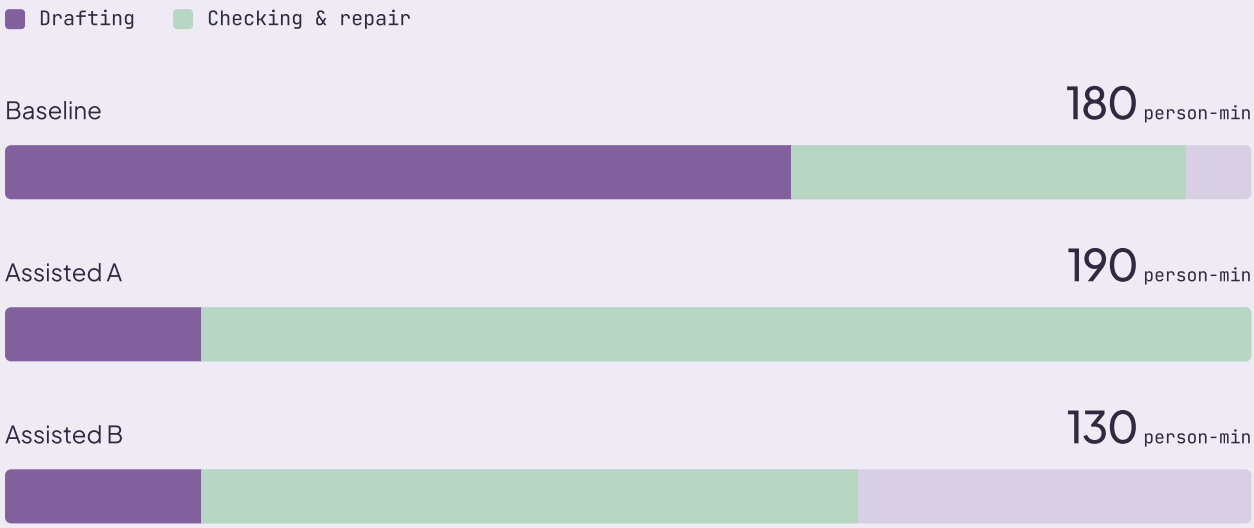
A third complication is rejection. If C never meets the acceptance contract, its 110 minutes still belong in the assisted workflow's cost, but it contributes no accepted change. Counting only merged pull requests would hide that failed attempt. Equally, if the baseline abandons a task and assistance completes it, the assisted workflow has delivered additional value that a comparison of completed-task times alone can miss.

To reproduce the arithmetic locally, sum each pair of columns. For the first assisted scenario, compute `100 * (190 / 180 - 1)`; for the revised scenario compute `100 * (130 / 180 - 1)`. The results are approximately `+5.56` and `-27.78`. These calculations verify the example, not the performance of any tool.

FIGURE 03 ILLUSTRATIVE ARITHMETIC · NOT MEASURED

Same drafting gain. Opposite outcomes.

Three fictional changes show why the cost of checking and repair belongs in the productivity calculation.



01

180 min

Baseline

120 drafting + 60 checking and repair.

02

190 min

Assisted scenario A

30 drafting + 160 checking and repair: 5.6% more total effort.

03

130 min

Assisted scenario B

30 drafting + 100 checking and repair: 27.8% less total effort.

Drafting falls 75% in both scenarios. Charge or explicitly amortize the cost of creating better contracts and fixtures in a real evaluation.

Original Kymata Labs worked example; all values are hypothetical.

Measure the workflow you intend to buy

A local evaluation should begin with a decision. Perhaps the team wants to authorize a tool for routine maintenance, expand its use into larger changes, or determine whether an existing subscription saves effort. The sample, acceptance endpoint, and cost boundary should match that decision. A demonstration chosen for its visual impact cannot answer all three.

Select tasks before seeing how the tool handles them. Preserve the task description, repository state, tool configuration, and acceptance contract. Include the task categories and experience levels to which the eventual decision will apply. Record exclusions and unfinished work. A convenience sample may still be useful, but its convenience should remain visible in the conclusion.

Where practical, assign eligible tasks to assisted or unassisted conditions before work begins. Avoid asking the same person to solve the identical task twice as the principal comparison: learning from the first attempt changes the second. If random assignment is impractical, describe the comparison as observational and consider competing explanations such as task mix, familiarity, or changes to the development environment. These are design recommendations, not a claim that one small team can eliminate every confounder.

Keep the acceptance bar stable. Record assistance actually used, since permission to use a tool and extensive use are different treatments. Include configuration, training, and measurement overhead where the purchase decision requires them. Report subscription, inference, and CI costs separately from person-hours, using an explicit conversion if combining them into money. Compare stable, similarly scoped task classes; a pull-request count does not assign equal value to different changes, and splitting one change into several should not inflate the result. Report both person-hours and elapsed time, together with accepted, abandoned, and still-open tasks. Show uncertainty and individual task outcomes when aggregate averages would conceal a few expensive failures.

Follow accepted changes for a declared period suited to the release cycle. Record attributable repair work and acknowledge uncertain attribution. An arbitrary short window cannot prove that later costs are zero; an indefinite window makes the evaluation impossible to finish. Report the window as a limitation, then continue ordinary operational monitoring.

Finally, publish the result in terms a buyer can use: this workflow, these tasks, these people, this tool configuration, this acceptance standard. Separate faster delivery from reduced effort and from improved scope. A tool may be worth adopting for one of those reasons even if another measure is unchanged. A negative result may justify narrowing the use case or improving the acceptance path before repeating the experiment, rather than declaring all assistance useless.

What this paper can and cannot establish

The empirical centerpiece is a reproduction, not a new trial. The source package contains METR's README, anonymized CSV, and regression code from a captured repository revision. The reproduction uses treatment-group mean imputation for missing post-review times, the published log-linear specification, and a normal 95% interval under the homoskedastic variance estimate. It recovers the repository's rounded headline result. The other variance specifications described in that repository were not independently reproduced for this paper. [1]

Only the inspected repository materials support our account of that study. The full paper and announcement were not retrieved in this research environment. Accordingly, this paper avoids claims about mechanisms, participant perceptions, tool-specific effects, or sensitivity analyses that would require those texts. A repository update in 2026 is not treated as a new 2026 productivity result.

The additional sources serve different jobs. Google's review documents supply explicit engineering criteria, not a productivity effect size. SWE-bench and GitHub Next supply inspectable evaluation boundaries, not estimates of organizational return. Four Keys supplies an example of measuring delivery

and failures, with its maintenance status disclosed. GitHub's responsible-use documentation supplies the vendor's own account of assistance and review responsibilities, not independent evidence of performance. [2][3][4][5][6][7]

No new security incidence rate, defect rate, industry adoption figure, or legal conclusion is inferred from this set. The queue mechanism, acceptance record, local evaluation design, and three-change calculation are this paper's analysis and proposals. They have not been validated as a combined intervention. They are offered in a form a team can challenge with its own observations.

The main unresolved empirical question is conditional: for which combinations of work, developer familiarity, tool configuration, and acceptance infrastructure do savings survive the complete workflow? This source set cannot estimate that surface. It can help a reader avoid substituting a generation demonstration or one historical average for the answer.

Keep the proof attached to the change

The useful ambition for AI coding is larger than producing more patches. It is making worthwhile changes easier to deliver and sustain. That ambition requires taking apparent speed seriously enough to measure where it goes.

For a team adopting assistance, the immediate action is modest: choose one bounded class of work, define acceptance before implementation, and retain effort and outcome evidence through that endpoint. Make rejected attempts visible. Keep reviewer labor separate from waiting time. Revisit the result after a declared follow-up window.

For tool builders, there is a corresponding opportunity. A system that makes its assumptions legible, keeps a change scoped, and connects evidence to the inspected revision can create value after the code has appeared. Generating a second candidate is useful only if the path to choosing and accepting it is affordable.

The title's claim should therefore remain contestable in every local setting. Sometimes code is still the expensive part. Sometimes the existing checks make proof comparatively cheap. But when generation becomes abundant, organizations need a way to tell valuable abundance from a queue of plausible obligations.

Keep the proof attached to the change, and a genuine saving has somewhere to show up.

Questions worth asking.

Does this mean AI coding tools make developers slower?

No general conclusion follows. The reproduced METR result concerns 16 experienced open-source developers, 246 issues, and early-2025 tools. It cannot estimate the effect of every current tool, task, or team. It is useful evidence that apparent generation speed should be checked against a declared completion boundary. [1]

If the tests pass, why is further review necessary?

Passing tests establish that the inspected implementation satisfies those tests in that environment. Acceptance may also depend on requirements the suite does not express, integration behavior, and maintainability. The appropriate additional review depends on the change. Google's guidance specifically asks whether tests would fail for broken code; GitHub warns that generated tests may miss scenarios. [3][7]

Can AI reduce the cost of verification itself?

Yes, that is a serious possibility to test. Documented uses include generating tests, suggesting fixes, explaining code, and summarizing changes. Those capabilities can contribute to a lower-effort workflow, but documentation alone does not establish the size of a saving. Include the cost of validating those outputs and apply the same acceptance standard to both conditions. [7]

What should a small team measure first?

Choose one bounded class of work and define what counts as accepted. Record author effort, reviewer and repair effort, elapsed time, and whether each attempt was accepted, abandoned, or remains open. Add a declared follow-up period for attributable repairs. This is a proposed starting method, not a validated universal productivity score.

Is the goal to produce less code?

The goal is to make worthwhile changes easier to deliver. Producing more code may be useful when it creates valuable capability at an acceptable cost. Producing less may be useful when a simpler change meets the same need. Output volume belongs beside evidence of acceptance and value; it cannot substitute for either.

SOURCES, NOT SCENERY

References.

1. **Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity**
   METR · 2025–07–10 · Primary research data and analysis code

   Official README, CSV, and regression.py captured at commit 7ff2d7670235f531dff77eff58355bd77392f396. Date is announcement date linked from README. 246 issues, 16 developers; early-2025 setting. Reproduced adjusted time effect 18.839994% with normal homoskedastic 95% CI 1.255985% to 39.477624%. Outcome is initial plus post-review implementation time, not all reviewer labor or lifetime maintenance. Full paper/announcement not retrieved. Sixteen CSV post-review values missing; published treatment-group mean imputation used.

   https://github.com/METR/Measuring-Early-2025-AI-on-Exp-OSS-Devs

2. **The Standard of Code Review**
   Google Engineering Practices · Undated; captured 2026–11–16 · Primary engineering guidance

   Favors changes improving overall code health even if imperfect; balances progress against quality. Normative engineering criteria, not measured AI productivity evidence. Captured full raw document.

   https://github.com/google/eng-practices/blob/master/review/reviewer/standard.md

3. **What to look for in a code review**
   Google Engineering Practices · Undated; captured 2026–11–16 · Primary engineering guidance

   Design, functionality, complexity, tests, context and maintenance criteria. Explicitly asks whether tests fail when code is broken. No AI defect-rate inference permitted.

   https://github.com/google/eng-practices/blob/master/review/reviewer/looking-for.md

4. **SWE-bench: Can Language Models Resolve Real-World GitHub Issues? — project README**
   SWE-bench authors and maintainers · Living repository; captured 2026–11–16 · Primary benchmark implementation documentation

   README defines repository-plus-issue to patch task and Docker evaluation workflow. No leaderboard score, pass rate, organizational return, or current model comparison claimed. Full README captured; underlying paper not inspected for this article.

   https://github.com/SWE-bench/SWE-bench

5. **Four Keys — project README**
   DORA team / Four Keys contributors · Undated; captured 2026–11–16 · Primary delivery-metrics implementation documentation

   Explicitly unmaintained at capture. Used only as an example of measuring delivery lead time and change failures, not current complete DORA definitions or recommended deployment software. Library release limitations disclosed in source; manuscript does not recommend using package.

   https://github.com/dora-team/fourkeys

6. **Copilot Timed Experiment**
   GitHub Next · 2022–03–03 repository commit; captured 2026–11–16 · Primary experiment artifact

   README, task, tests, and history captured at commit 8ec9e3916cbff3bed32e68e68042e08121989b7a. Timer ends at complete passing GitHub Classroom test suite. Toy HTTP parser/response engine excludes actual network requests. No effect size, participant sample, or full report in inspected artifact; exact linkage to published 2023 study not independently established. No numerical gain cited.

   https://github.com/githubnext/copilot-timed-experiment

7. **Application card: GitHub Copilot Chat**
   GitHub · Source revision 2026–10–28; captured 2026–11–16 · Vendor product capability and responsible-use documentation

   Official documentation source file. Documents tests, fixes, explanations, and PR summaries, while requiring review and warning generated tests may not cover all scenarios. Capabilities are vendor-described intended uses, not independent measured savings. Atom file history captured; revision date is not a launch date. Rendered docs site not fetched.

   https://github.com/github/docs/blob/main/content/copilot/responsible-use/chat.md