// WPR-CLD-005 · PUBLISHED 15 NOVEMBER 2026

What Breaks When AI Agents Ship Software at Volume: Wolves at the Gate

Six bottlenecks from five months of measured records: the fixes we tried and which of them hold, where each fix leaks, and how to tell a real benefit from a cost that moved somewhere you stopped looking

Jason Newell · MAX Research Collective · Report WPR-CLD-005 · Published 15 November 2026
Licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/
Original: https://www.maxresearchcollective.com/articles/what-breaks-when-ai-agents-ship-software.html
Cite as: Newell, J. (2026). What Breaks When AI Agents Ship Software at Volume: Wolves at the Gate. MAX Research Collective, WPR-CLD-005.
https://www.maxresearchcollective.com/articles/what-breaks-when-ai-agents-ship-software.html

In short

Between June and mid-November 2026, one developer and a team of AI agents opened 3,332 pull requests across nine repositories and merged 3,222 of them. Those pull requests ran 33,799 workflow runs of CI (continuous integration), the automatic build and test checks that run on every change before it merges. Writing the code stopped being the hard part early on. Every bottleneck we hit was downstream of the code:

- CI minutes and runner queues;
- review capacity;
- tests that couldn't fail;
- instruction files too large for the tools that read them;
- a bill nobody was metering whole.

Most of those problems can be solved. Every solution we tried also leaked somewhere. Some benefits held up under audit. Others turned out to be the same cost, moved to a meter we weren't reading.

We ended up with a short test for telling a real benefit from a wolf in sheep's clothing. A real benefit names the meter it saves on, meaning the count its cost shows up in, such as runner minutes, dollars, tokens or hours of someone's attention. It's measured where the cost lands, not where it starts. It can be shown to fail. And it still holds a month later.

Table 1. The six bottlenecks, the fixes that hold, and where each fix leaks.

BOTTLENECK	WHAT HOLDS	WHERE IT LEAKS	VERDICT
CI minutes and queues	Fewer schedules and triggers, job timeouts, dependency caching	Self-hosting cut the bill from about $108 to $24.74 a month and raised the waits	The cuts are clear wins. Self-hosting hasn't earned that yet.
AI review	CodeRabbit at a flat seat price	Silent fallback to default settings, skipped reviews, and Copilot answering 7% of requests	CodeRabbit is a net gain. Copilot came out at zero or worse.
Tests that can't fail	Guards for whole classes of defect, and a SAST ratchet for hygiene	Mutation testing mostly guards itself (283 of 474 mutations), and scanners miss logic defects	Guards are a clear gain. Mutation testing is mixed.
Instruction files and ledgers	Proposed, not yet measured: an index for each reader and a size budget	The files outgrow the readers they brief	Valuable, and a cost moved into context until it has a budget.
The whole bill	Caps on every metered line, which took $912 to $262	The agent line had no cap and grew back to a projected $917	Caps work only where they're set.
Parallel sessions	One pull request at a time where generated files get rebuilt	Serializing the work serializes the waiting	A gain for generated files, a cost for independent work.

The bottleneck moved

When code costs almost nothing to write, it piles up wherever it has to be checked.

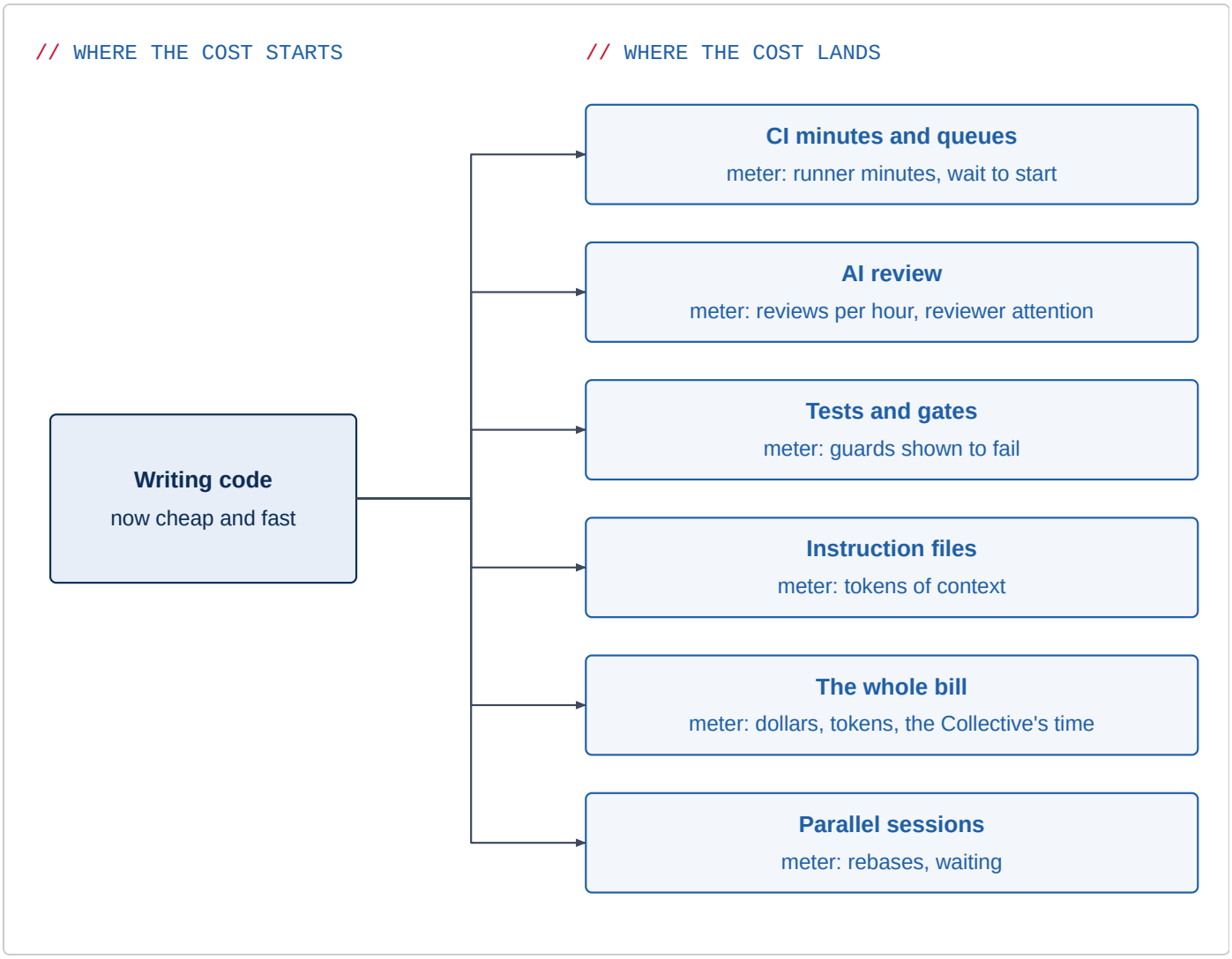


Figure 1. Code is cheap to write, so the cost lands downstream, each place under its own meter.

Faros measured this across 22,000 developers in 4,000 teams in April 2026:

- Task completion per developer rose 33.7%.
- Median time in review rose 441.5%.
- 31.3% more pull requests merged with no review at all.
- Bugs per developer rose 54%, and incidents per pull request more than tripled.

Faros is a vendor and the data is observational, so it can show what moves together and can't show what causes what. Our records show the same shape at the scale of one person.

In the first thirteen days of November, our repos opened 241 pull requests that ran CI. They used 2,285 workflow runs, 9.5 per pull request, and 2,764 minutes went to runs a newer commit cancelled. The agents moved fast and everything they handed off to ran behind.

The randomized evidence on whether AI makes an individual developer faster is unsettled. METR's 2025 trial found experienced developers took 19% longer with AI, while they believed they'd been about 20% faster. METR's February 2026 update estimated that its returning developers took about 18% less time, with an interval that crosses zero, and METR calls its own measure "likely a bad proxy." We don't lean on either. The question here is narrower: given that the agents produce work at this rate, what does it take to check it, and what does the checking cost?

CI minutes run out, and self-hosting trades the bill for the queue

The problem. GitHub Actions, the service that runs CI, includes a monthly allowance of compute: on Pro, $18 a month, which at the 2-core Linux rate of $0.006 a minute buys 3,000 minutes on the machine that runs each job, the runner. Parallel agent sessions spend it in days.

- In July, one repository used about 90% of the month's minutes in three days. A test pyramid of roughly 50 job-minutes ran on every push to every branch, from several Claude sessions at once.
- In October, the 3,000 included minutes were gone by the fourth day.
- In the first week of September, one repo measured 295 runs and about 925 billable minutes. That included 19 cancelled runs, each billed for the time it ran.

What we did. Two things, and the cheaper one did more.

The cheap fixes cut runs that answered no question. Each is a clean saving, because the run it removed produced nothing.

- A chat-learning refresh on a two-hour schedule was 83 runs a week, more than that repo had pushes. It became manual-only.

- An AI review lane, meaning one automated reviewer and the workflow that runs it, fired on every new commit, about 1,200 runner-minutes a month. It now fires only when a pull request is opened or marked ready.
- Seven jobs had no timeout, and one hung job billed for 24 hours. Every job now has one.
- A deploy that uploaded files one at a time took 20 to 33 minutes. It became one mirror command, and the whole deploy now takes about 64 seconds.
- Dependency caching took one repo's install from about 20 minutes to under one.

The larger change was moving CI onto the Collective's own PC as self-hosted runners, machines you own that run the same jobs and whose minutes GitHub doesn't bill. Our monthly GitHub bill fell from about $108 to $24.74, and so far in November it's $0.00.

Where it leaks. The bill went down and the waiting went up.

- In the timed set in our earlier piece, The Whole Bill, a hosted job waited 6 seconds on average to start. A local job waited 15.4 minutes.
- November has been better. From the 1st to the 13th, jobs waited 1.8 minutes on average on the shared runners and 4.9 minutes on the dedicated ones. Hosted jobs waited 1 second.

The averages hide the incidents, none of which can happen on hosted runners:

- Three jobs waited 13 hours while their runner was offline.
- On 13 November, GitHub still listed one runner as online while its host had stopped.
- On 2 November, a job waited 627 minutes while two lanes and a flex runner stood free.
- On 31 October, the PC's Linux subsystem stopped answering for nine hours, and CI on main waited for all of it.

One PC is one failure domain.

Jobs also ran slower: 4.3 minutes on average locally against 1.6 hosted. Most of that gap comes from jobs contending for the same machine, and a repository with the runners to itself ran in 3.2 minutes against 3.0 hosted.

The slowdown broke every time limit set for the hosted rate. A mutation-testing job, which plants bugs in the code and checks that the tests catch them, had a 75-minute timeout sized for hosted speed, where each mutation took at least 8.7 seconds. On the PC each took at least 24.6. The job was cancelled at 75 minutes with 173 of its 319 mutations done.

Then there's the controller, the program that decides which job runs where when nine repos share one PC. Ours took 413 pull requests to build, 315 of them in October. At one point the controller made 11,944 GitHub API calls in thirty minutes, against a limit of 5,000 an hour. The API stopped answering, and one pull request's CI never ran.

GitHub's pricing isn't settled either. It announced a $0.002-a-minute platform charge on self-hosted runners for March 2026. It then postponed the charge "to re-evaluate our approach," and hasn't cancelled it.

Other mitigations.

- Keep a small hosted budget for overflow. When a job has waited past a threshold, send it to a hosted runner. That caps the worst wait for a few dollars.
- Measure time from pull request opened to green. Minutes billed are what the vendor charges; time to green is what the work pays.
- Size every ceiling on the rate where the job actually runs. Split the work when no ceiling fits.
- Call a runner healthy only when its host answers. GitHub's "online" listing can outlive the host.

Verdict. The cash saving is real but small. It's about $83 a month, or roughly $63 to $76 net of electricity; we haven't metered the power, so that range is an estimate. The trigger and schedule cuts, timeouts, caching and the single-command upload are clear wins with nothing hidden. Self-hosting hasn't earned that yet. Count the waits, the pull requests that built the controller, and the agent tokens spent waiting on runners, and the net may be negative. We haven't measured enough to say which way it falls.

AI review cannot keep pace, and "the review ran" is not "the review looked"

The problem. Every code-review service we tested is priced for human-speed work.

- CodeRabbit sells a seat with an hourly allowance: 8 reviews an hour per developer on its Team plan, which may adjust under ”sustained high-volume” use. From first use to mid-October we made 1,322 requests. It answered 749 (57%) and refused 573. Once the allowance ran out, 2 of the next 127 requests ran. In August its queue reached about 36.6 hours.
- Copilot’s code review answered 60 of 822 requests (7%). Since June it bills twice, in AI credits and in Actions minutes, so a review that never arrives can still cost.

What we did.

- More lanes. We ran two reviewers whose findings barely overlap: in a July bake-off, Gemini and Qodo shared about 2% of 394 hand-rated findings on 110 pull requests.
- A throttle-aware request tool. It reads each reviewer’s own ”available in N minutes” notice and waits or skips.
- A budget switch for Copilot. Our usage controller holds it, and it’s been off since November. The refusals stopped.
- A local reviewer. A reviewer model runs on the PC, with no per-review price.
- A rule. Since 20 October, a review that hasn’t landed when everything else is green is skipped, and the pull request merges.

Where it leaks.

- Silence that reads as approval. Our CodeRabbit instructions grew 159 characters past the vendor’s 20,000-character limit. CodeRabbit rejected our configuration and reviewed with its default settings. The reviews kept arriving and looked normal, and none applied our rules. Twice, a rate-limited review also posted a green ”success” status. A reviewer that degrades in silence is worse than one that refuses, because a refusal at least gets noticed.
- Prompt ceilings. Copilot’s review prompt is capped at 110,000 tokens; its job log reports MaxPromptTokens=110000. Our main instruction file reached 365 KB,

roughly 90,000 tokens, which is about 80% of that cap before a line of the diff. Every Copilot review failed with "Prompt too big," documentation-only pull requests included.

- Skipped by design. The rule against waiting buys throughput with coverage. Between 1 and 13 November, 46 CodeRabbit requests came back "could not review," and nothing records any of them being redone. On one repo the median pull request lived 7.2 minutes from open to merge, which is less time than a review takes to arrive.
- "Included" that was not. The Claude review lane was listed as included in the plan. It ran $0.70 to $2.34 a run in API cost. On one bad day, ten runs at about $2.34 each hit their turn limit and produced one usable review. We paused it.
- A precision that cannot fall. The local reviewer's ledger reports a precision of 1.0, meaning every judged finding was real. It holds 2,248 findings, of which 82 (3.6%) have been judged. All 82 were confirmed and none refuted, because the adjudicator was never built to write "refuted." A score that can't go down carries no information.
- Fixes that need reviewing too. More than once, a reviewer found a new defect inside the fix for the last one. One reviewer marked threads "□ Addressed" on commits that hadn't fixed them.

The published research agrees that most of what AI reviewers say is rejected. A July 2026 study of CodeRabbit in the wild covered 31,073 pairs of CodeRabbit reviews and developer responses from 10,191 pull requests in 239 repositories. 36.4% were accepted, 7.3% led to discussion and 56.3% were rejected, mainly as invalid suggestions: incorrect flags, redundant comments or out-of-scope remarks.

What reviewers caught that our tests did not. This is the case for keeping them. In our site's findings register, 147 of 368 defects name a hosted reviewer as the finder. That's our own, loose count. On another repo, a deterministic linter replayed against 163 things the reviewers had flagged caught 7. Reviewers find what rules don't. What no record of ours checks is the reverse: would the test suites, run on the commit before the fix, have caught the defect anyway? Until someone runs that, "reviewers caught X" is an upper bound.

Other mitigations.

- Route review by risk. Ask for it on the paths where a defect costs most: security, money, mail and data. Skip it where the deterministic suites already grade the change.
- Keep diffs reviewable. Faros's 2025 report tied AI adoption to a 154% rise in average pull-request size. A review allowance counts reviews and ignores lines, and a reviewer reads a 50-file diff no better than a person does.
- Prove the reviewer read your configuration. Read the vendor's limit, print the headroom on every change, and fail the build before the file crosses it.
- Grade each reviewer on a fixed sample. Mark each finding real or refuted, so its precision can actually fall.
- Count a skipped review as a coverage gap. Record it, and redo it on the risky paths.

Verdict.

- CodeRabbit at a flat seat price is a net gain. It's the one lane with sustained output, and its findings fill our register.
- "Free" Gemini was partly a wolf. It was free per review, but cost about 1,200 Actions minutes a month until we narrowed when it fires.
- Copilot came out at zero or worse. Its allowance is shared monthly across every repo, and at agent volume it ran out.
- A local reviewer with no price still has a cost, paid in the PC's memory and runner time. Its value is unmeasured.

Tests that cannot fail, and gates that pass what they should catch

The problem. A check that can't fail looks exactly like a check that passed.

- One of our gates printed GATE fails=1 and exited 0. It survived three separate fixes before every layer around it was driven until it failed.
- Four of six clauses in another guard compared 0 == 0.
- On another repo, five review fixes shipped with no test, and the mutation probes caught all five.

- On a third, one anti-pattern recurred eight times in a day, five of them after the rule against it was written.

The security scanners, known as SAST (static application security testing, which reads source code for vulnerability patterns), had the mirror-image problem. In August an audit ran Bandit, Semgrep, Ruff and ESLint over our mail endpoints, and all four passed two real defects:

1. A sending loop paced on messages accepted instead of recipients attempted.
2. Twelve sign-up requests could send twelve emails to an address that wasn't the requester's.

A manual attack found both.

What we did.

- Mutation testing. It deliberately breaks the code, one planted bug at a time, and requires a test to notice each one. Our site's harness grew from 21 mutations in August to 474. Another repo's grew from 55 probes to 2,014, and on 11 October all 1,819 run were caught.
- Guards for whole classes of defect. These cover every instance of a defect type at once, and they found live defects the day they were written. Three stated counts were stale. A folder of photos and a video were still public on the web host months after we dropped them from the site. And the deploy checked for "at least 80" article pages while the repository held 86.
- A SAST baseline with a ratchet. The baseline records the findings already accepted, and the ratchet lets that count fall and never rise. One repo started with 135 Bandit findings, 85 of them try/except/pass. Of those 85, eight were hiding a real failure. The baseline came down to 43, with no high-severity findings.

Where it leaks.

Mutation testing mostly tests itself. Of our 474 mutations, 283 (59.7% by count, 60.6% by weight) guard the checking machinery: the test suites, CI and the guidance files. Only 35 guard what a reader touches, the server endpoints and the research assistant. Most of the defects the mutation harness found were in the harness.

Mutation runs moved off pull requests to a weekly sweep on 21 October. Since then, 25 new findings have been registered:

Table 2. Findings registered since 21 October, by who found them.

FOUND BY	FINDINGS
The Collective	7
A reviewer	6
An article build	6
The deploy, CI or telemetry	6
A mutation sweep	0

It is slow where it runs. The weekly sweep was cancelled at 75 minutes: its four shards queued for two runner places, and the last two never finished. The ceiling is now 300 minutes, and the last full run took 2 hours 45 minutes.

It leaves a week-long window. With no mutation run on pull requests, a guard weakened on Monday isn't proven until the next weekly run, up to seven days later. One probe survived on the CI runner while it was killed on the development box. A local run proves less than it appears to.

Scanners grade patterns and miss intent. Neither defect SAST missed is a pattern. Both are logic about what a control counts. The published numbers point the same way:

- A 2023 study of Java tools found they detected 12.7% of real-world vulnerabilities; even combined, they missed 70.9%. Yet the same tools "perform well on synthetic benchmarks."
- A 2024 study of C and C++ found a single tool flagged vulnerable functions in 52% of the commits that introduced vulnerabilities, and at least 76% of its warnings in those functions were irrelevant to those commits.

Other mitigations.

- Mutate the diff. Google mutates changed lines only and suppresses mutants on code that doesn't matter, such as logging. That raised the share of useful mutants from 15% to 89%. In its study of 1,502 high-priority bugs, 70% were coupled to a mutant that could have prevented them. Mutants make good guidance; as a score, the evidence that they predict real fault detection is mixed.
- Weight the mutation budget by where a defect costs most. Our current weight follows where the test apparatus is most complex.
- For logic and abuse defects, write attack tests that count effects. Our mail suite now counts the messages that actually leave instead of the records the code happens to write. Counting effects is what exposed both defects SAST missed.
- Fault-inject every new guard before it ships. Break the thing it guards, watch the specific clause go red, and restore it. A green suite after an injection proves nothing until the injection is shown to reach the guard.

Verdict.

- Guards for whole classes of defect are a clear net gain. They found live defects on their first run.
- The SAST ratchet is a gain for hygiene and no help on logic.
- Mutation testing is mixed, and partly a cost that moved. It proves guards can fail, which matters. But as we run it, most of its weight and most of its findings concern itself. Since it went weekly it has caught nothing that shipped, and it costs hours of PC time a week. The test is to count guards that were weakened, shipped, and then caught by a sweep. So far that count is zero.

The ledgers grow into the windows they were meant to fill

The problem. Every defect we fix leaves a lesson, and the lessons go into files the agents and reviewers read: a findings register, a known-failure-patterns ledger, the rules files. That's how the system learns, and those files grow without limit.

- Our site's register went from 59 findings on 6 August to 368.
- One repo's known-failure ledger is 978,379 bytes; another's is 1,268,328.

- The site's main agent instruction file is 258,574 bytes, roughly 65,000 tokens at four bytes a token.
- Local logs put the context carried by each agent request at about 499,000 tokens in November, up from 272,000 in October.

Where it leaks. The learning breaks the readers it was written for.

- Copilot's review failed outright once the instruction file reached 80% of its prompt cap.
- CodeRabbit dropped to default settings once its rules crossed 20,000 characters.
- A Claude review lane spent three to four of its twelve turns paging through the registry before it reached the diff.

Even under every limit, a longer context isn't free. Chroma tested 18 models in 2025 and found performance "grows increasingly unreliable as input length grows," even on simple tasks. Anthropic's own guidance describes an "attention budget" that long contexts spend. Carving the files down didn't hold either: one instruction file went from 365 KB to about 164 KB and is back at 222 KB.

Other mitigations.

- Compile the ledger into an index, and hand each reader the index. One repo condensed 230 failure classes into 35,766 characters. The full entries stay one link away for whoever needs them.
- Route the rules by path. A change to the mail endpoints needs the mail rules and can skip the figure rulebook. We already generate a "which files govern my change" table; feeding a reviewer only those files is the next step.
- Print the headroom on every change, and fail before the limit. The useful question is how much room is left before the next lesson can't be added.
- Retire lessons a guard now enforces. Once a mutation proves a check fails when the defect returns, the prose about that defect can move to history.

Verdict. The ledger has value: it's where most of our guards for whole classes of defect came from. Grown without a budget, though, it's a cost moved into context, and it degrades the reviewers it briefs. The fix is a budget.

Nobody meters the whole bill

The problem. The agent is the largest line on the bill and the worst metered.

- In August the stack ran about $912 a month. Capping every overage brought it to $262.
- The cap didn't hold for the agent. By 13 November, extra usage on Claude for the month stood at $305.48 or more, with no budget set. The month's total was $517.48 or more, projecting to about $917.
- That projection lands next to August's $912 and well above the $302.74 month we published in October.
- 88% of November's agent spend is on the largest model.

Where it leaks. The meters can't see each other.

- The local usage logs counted $23.20 for a week in which month-to-date extra usage passed $305. They don't see cloud sessions or other devices.
- Nothing in our telemetry meters total tokens. Every bytes-to-tokens figure in this piece is an estimate at about four bytes a token.
- The Collective's attention has no meter at all, though it shows up in the records: an idle, green pull request checked eleven times in a row; requests from a partner repository left 35 hours unanswered; "about 10 minutes of real article work became hours."

Mitigations.

- Set a budget on every metered line, the agent's included. The cut from $912 to $262 came entirely from caps, and the one line without a cap is the one that grew back.
- Tag spend by kind of task. Then "the CI controller cost X in tokens" becomes a measurement.
- Route routine work to a cheaper model. Our ledger estimated $41 to $109 a month saved by tiering. One rule, though, sent all building to the largest model to control token spend. Measure which was right before choosing again.

- Log the Collective's interventions (force-merges, rulings, restarts), and count them per week.

Parallel sessions collide

The problem. Three pull requests were open at once, from three sessions on two bases. Thirteen non-generated files were touched by more than one, and four by all three. On another repo, an ordinary day with several open pull requests cost a dozen rebases, and merge-conflict markers reached main inside a data file.

What we did. One pull request at a time per repository. New work goes into the open one.

Where it leaks. Serializing the work serializes the waiting. A documentation-only pull request spent about 90 minutes in a sweep, and a green one sat idle for 35 minutes behind another. The rule has been relaxed twice: two non-conflicting code pull requests may now run together, and documentation takes a faster lane.

Other mitigations.

- Give each path one owning session.
- Regenerate derived files after a merge instead of merging them.
- Check for conflicts against the open pull requests before every push.

Verdict. One pull request at a time is a gain for anything that rebuilds generated files, where a collision costs a full rebuild and a full re-check. For independent work it costs throughput, and the relaxations were the right correction.

The benefits, audited

A benefit gets through the gate when it names the meter it saves on, is measured where the cost lands, can be shown to fail, and holds a month later.

Measured, with nothing hidden:

- The single-command deploy: 20 to 33 minutes became about 64 seconds.

- Schedule and trigger cuts: 83 runs a week of a refresh nobody needed, and about 1,200 minutes a month of a review lane firing on every commit.
- A timeout on every job.
- Dependency caching: one repo's install went from about 20 minutes to under one.
- Guards for whole classes of defect, which found live defects on their first run.
- Overage caps: $912 down to $262, for as long as every line had one.
- One pull request at a time, where a collision means a full rebuild.
- CodeRabbit at a flat seat price.

Wolves, or wolves partly tamed:

- "Free" review that billed in another currency. Gemini cost Actions minutes, and Copilot now costs Actions minutes as well as credits.
- "Included" review that cost $2.34 a run and mostly posted nothing.
- Self-hosted savings reported in billed dollars, with the queue, the controller and the waiting agents left off the ledger.
- A precision of 1.0 with no way to record a miss.
- A mutation suite that reads as safety while 60% of its weight guards itself.
- A ledger that "learns" until it no longer fits in the reviewers it briefs.
- Pull-request counts as velocity. 3,222 merged pull requests in about five and a half months is a real number, but here's what sits inside it:
- In November, 49 of 241 pull requests were documentation only.
- 393 of them built the CI controller.
- The 1,582 pull requests of one flagship app ended with that app frozen and replaced by a new one, built as "not a port."
- On that app, half of a sample of 56 merged pull requests needed more than one round to pass.

Throughput measures activity. Value delivered has to be counted separately: releases, features a person uses, defects that never reached anyone.

None of these were bad decisions. Each was the right next move on the evidence of the day. The pattern is that each one moved a cost to a meter nobody was reading yet.

What would settle the open questions

We don't have these measurements yet, and we're adding them before anything else:

1. Time from pull request opened to green CI, per pull request, on the same repositories: hosted in September against local in November.
2. Agent spend tagged by task: product, CI and controller, documentation, review handling.
3. For every defect credited to a reviewer, the test suites run on the commit before the fix, to see whether the reviewer was the only catch.
4. Weekly mutation results counted as guards that were weakened, shipped and then caught, rather than as mutations killed.
5. Merged pull requests classified by the paths they change: product code, rules and documentation, CI and infrastructure.
6. The Collective's interventions, counted per week.

Where our own records disagree

Some of our figures disagree with each other. We've left them out of the argument until they're settled:

- November's hosted minutes. The CI ledger says 2,765; GitHub's billing page says 1,476, in the same snapshot.
- The October day the minutes ran out. One record says all 3,000 were used and $6.06 was billed. Another says 2,015 were used and nothing was billed.
- CodeRabbit's hourly allowance. At different times it read 10, 8, 5, 4, 2 and 1.

Where a figure here has a date or a window, it's the one the record gives.

Why it matters

Agents make code the cheapest part of a software project, so the work around it sets the pace: the machines that test it, the reviewers that read it, the files that brief the next session, and the bill. A team that adopts a fix without naming the meter it saves on will report savings that a different meter quietly absorbs, and the cost lands on whoever reads that meter last: the developer waiting on a runner, the reader who trusts a review that never ran, the person who opens the invoice at the end of the month. Anyone running agents at this volume should expect every fix to move a cost, and should find out where before calling it a win.

The open questions above are meters nobody has read yet. Until someone reads them, every benefit in this piece that we haven't measured is a claim about where the wolf went.

Glossary

TERM	DEFINITION
CI (continuous integration)	The automatic build and test checks that run on every change before it merges.
Context window	How much text, counted in tokens, a model reads at once.
Mutation testing	Planting small bugs in code, one at a time, and requiring a test to notice each one.
Ratchet	A limit that can tighten and never loosen, so a count that has fallen can’t rise again.
Review lane	One automated reviewer and the workflow that triggers it, such as the Claude review or CodeRabbit.
Runner	The machine that runs a CI job. A self-hosted runner is one you own, and GitHub doesn’t bill its minutes.
SAST (static application security testing)	Scanners that read source code for known vulnerability patterns without running it.
Token	The unit a model reads and bills by, roughly four bytes of text in the estimates here.

Sources

- Faros, ”Acceleration Whiplash,” 2026-04-27, https://www.faros.ai/blog/ai-engineers-faster-delivery-slower ; ”The AI Productivity Paradox,” 2025-07-23, https://www.faros.ai/blog/ai-productivity-paradox-report-2025
- METR, early-2025 experienced developer study, 2025-07-10, https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/ ; uplift update, 2026-02-24, https://metr.org/blog/2026-02-24-uplift-update/

- Lin, Liang, Thongtanunam and Tantithamthavorn, CodeRabbit reviews in the wild, arXiv 2607.03316, July 2026, https://arxiv.org/abs/2607.03316
- GitHub Docs, Actions billing, https://docs.github.com/en/billing/concepts/product-billing/github-actions , and limits, https://docs.github.com/en/actions/reference/limits , read 2026-11-13; GitHub, "Pricing changes for GitHub Actions," 2025-12-15, https://github.com/resources/insights/2026-pricing-changes-for-github-actions ; GitHub changelog, Copilot code review will consume Actions minutes, 2026-04-27, https://github.blog/changelog/2026-04-27-github-copilot-code-review-will-start-consuming-github-actions-minutes-on-june-1-2026/
- CodeRabbit, plans and fair use, read 2026-11-13, https://docs.coderabbit.ai/management/plans ; configuration directory limits (instruction body, 20,000 characters; our own instruction block was rejected at 20,159), https://docs.coderabbit.ai/configuration/configuration-directory
- Copilot code review errors "Prompt too big after adding system message" and "MaxPromptTokens=110000," from our CI logs; the same errors are quoted in public repositories, for example https://github.com/maccoss/einzel/pull/54
- Petrović, Ivanković, Fraser and Just, "Practical Mutation Testing at Scale," IEEE TSE 2021, https://arxiv.org/abs/2102.11378 ; "Does mutation testing improve testing practices?," ICSE 2021, https://ar5iv.labs.arxiv.org/html/2103.07189
- Just et al., "Are Mutants a Valid Substitute for Real Faults?," FSE 2014, https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf ; Papadakis et al., ICSE 2018
- Li et al., SAST tools for Java, ESEC/FSE 2023, https://marklee131.github.io/Java_CVE_Bench ; Charoenwet et al., SAST in secure code review, ISSTA 2024, https://arxiv.org/abs/2407.12241
- Chroma, "Context Rot," 2025-07-14, https://www.trychroma.com/research/context-rot ; Anthropic, "Effective context engineering for AI agents," 2025-09-29, https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents
- Our own records: GitHub API counts to 2026-11-13; our CI and spend telemetry of 2026-11-13; the findings registers of our repositories; and, on this site, What Building with AI Really Costs: The Whole Bill and False Green.

CLOUD / DEVOPS WPR-CLD-005

// ABOUT THE AUTHOR

Jason Newell leads organizations and people in technology, and is a researcher, deep technologist and builder — from a CRAY to agentic AI. He founded MAX Research Collective to build AI for regulated industries that runs where its owner has control, with healthcare as its strongest focus. About Jason → Connect on LinkedIn → ORCID →