// ART-CLD-013 · PUBLISHED 23 NOVEMBER 2026

Building Software With AI Agents: The Rule That Stayed Text

A written rule stops nothing until a machine enforces it, and a process run by AI agents shows which of its rules were never more than words.

Jason Newell · MAX Research Collective · Article ART-CLD-013 · Published 23 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/building-software-with-ai-agents.html
Cite as: Newell, J. (2026). Building Software With AI Agents: The Rule That Stayed Text. MAX Research Collective, ART-CLD-013. https://www.maxresearchcollective.com/articles/building-software-with-ai-agents.html

The path in one paragraph

Software built with AI agents moves from instruction to production through eight stages: a written instruction, a design, a build on a private branch, tests, a ledger of past failures, two passes of review, continuous integration, and a release. Each stage has a rule, and the rules that survive are the ones a machine checks. The rule that stayed text longest in this process was the one that mattered most: a change merges only when its checks are green. Nothing stopped a merge over red checks except a sentence in a file — and the record shows what a sentence is worth when someone is in a hurry.

The process belongs to one developer and four repositories worked by AI agents. Figure 1 shows the whole path. It reads like a snake: left to right along the top row, back along the second, then forward again through the release, with one way back to the build for every change that fails the merge gate.

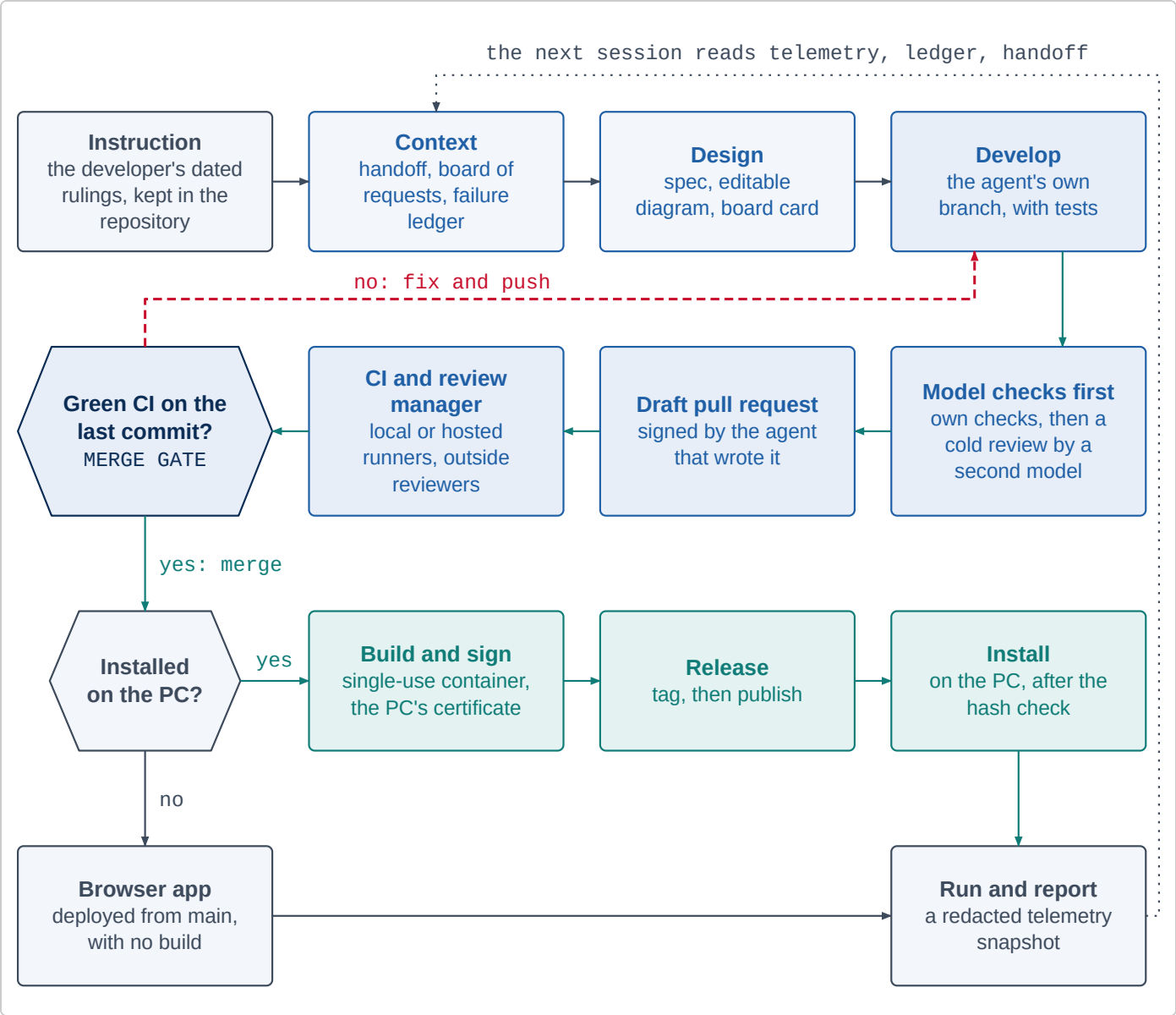


Figure 1. Follow the path from the top left: the dashed red line is the only way back from the merge gate, and the dotted line carries what the running app reports to the start of the next session.

The fleet

Four repositories follow one process. A web application is the oldest, and it's frozen: it takes fixes but no new work. A Windows desktop application receives all new work. A voice and text assistant is the newest, three days old at the time of this review. The fourth repository plays two roles. It's a product with users of its own, and it's also the control service, the program that places continuous integration jobs on machines, asks

the outside reviewers for opinions, builds the desktop installers and collects telemetry for the other three.

AI agents are the builders. Sessions of one coding agent do most of the work, a second coding agent works through the control service, and a third takes small issues on the code host. The developer sets every rule and owns every pull request. One personal computer is the production environment: it runs the applications, the CI machines and the build containers, which is why so many rules exist to protect its processor, memory and disk.

Stage one: the instruction

Every change begins as the developer's own words, kept in the repository with a date and a quotation mark so that any agent on any day reads the same rule. A shared entry file holds the rules every agent reads first, the routing table that says which documents govern which paths, and who signs and approves what. Each agent tool has its own small pointer file that sends it back to the shared one. A standing-rules file records the product and process rules with the exact words the developer used, and a newer ruling names the older one it overrides while the older text stays behind, marked as history.

Three mechanisms keep an instruction from drifting. The first is the dated quotation. The second is a checker that compares the same text across repositories: a script fails when the pointer files disagree, and a second one compares the block of rules that must be byte-identical everywhere. The third is computed routing. A small tool takes a list of changed files and prints which documents apply, so a change to a workflow file routes to the CI rules and a change to the native client routes to the user-interface specifications. Nobody has to remember where a rule lives.

A session gets its context in a fixed order. It confirms the branch and the working tree, and work happens on the session's own branch, never on the main branch. It reads the handoff file, which says where the last session stopped and what the exact next step is. It reads the board of requested items and the decision log. It reads the ledger of failure classes before designing anything and again before pushing. Last, it reads the live state: a start-of-session hook prints a telemetry digest from the production

machine, and alarms in that digest are work, done before anything else. The request itself goes on the board before any code exists, so it survives the end of the session and the compaction of its context.

Stage two: the design

A change is designed in three linked artifacts: a board card that records the ask, a Markdown specification an agent can build from, and an editable diagram file for each picture. A check fails the build when they aren't linked. The specification states scope, affected components, interfaces and data flows, expected behavior, failure cases, implementation steps and acceptance tests, and it carries sample code. A spec that an agent can't implement and verify from the text alone isn't finished.

Two rules shape most of the design work. Every change documents itself: it updates its own spec, its own diagram and every other spec or diagram it touches, and it cross-links them in the same pull request. And all research is stored, with each image as given, the text read from it, the source and the session's notes kept apart. A replaced document moves to a deprecated folder, and nothing live links into it.

A documentation checker fails the build when a picture has no editable source, when a spec has no declared kind or no link to another document, or when a relative link is broken. No cloud session can check how a screen looks, so the control service boots a Windows virtual machine in a container, installs the build and photographs each screen.

Stage three: the build

An agent builds one unit of work on its own branch, in its own cloud machine, and signs everything it produces so the developer can tell whose work it is. Every commit carries the developer's identity and a trailer line naming the agent. Every pull request is opened under the developer's account as a draft, and the first line of its body, and of every comment, names the agent that wrote it. The line is a signature and never an approval, and a script that's the same file in every repository checks all three.

Limits are written into each agent's role. A coding-agent session never commits to the main branch, never builds on the production machine and changes the CI runners only

through the control service. Subagents never merge. The second coding agent never pushes or posts for itself, and the control service relays everything it does. Hooks refuse a spin-off session without a written brief, and no session can end while it holds uncommitted work.

A change passes through a fixed loop:

1. Read the governing documents for the files to be touched.
2. Write or extend the tests with the change, and for a bug write the failing test first.
3. Run the repository's gate in the session machine: type check, lint, tests, and a compile that proves the change builds.
4. Have the diff reviewed cold by a second model that sees only the change and not the builder's reasoning, and answer every finding.
5. Bump the revision one patch step for anything other than documentation.
6. Check the working tree, restore any tracked file that wasn't meant to be deleted, and stage files by name, never by wildcard.
7. Walk the diff against the ledger and the ship checklist, then push once.

Done means pushed; the report names the branch and the commit, and a result that exists only in a container doesn't count.

Stage four: testing

Testing runs in four stages, and Figure 2 sets them side by side. The agent that writes a change writes the tests, runs them, and has the change reviewed cold before the push. Then the control service runs them again on local or hosted machines, scoped on a draft and in full once the pull request is marked ready. Its review manager decides whether the change may merge. And after the merge, only for a desktop application installed on the production machine, the build is signed, installed and checked on real Windows.

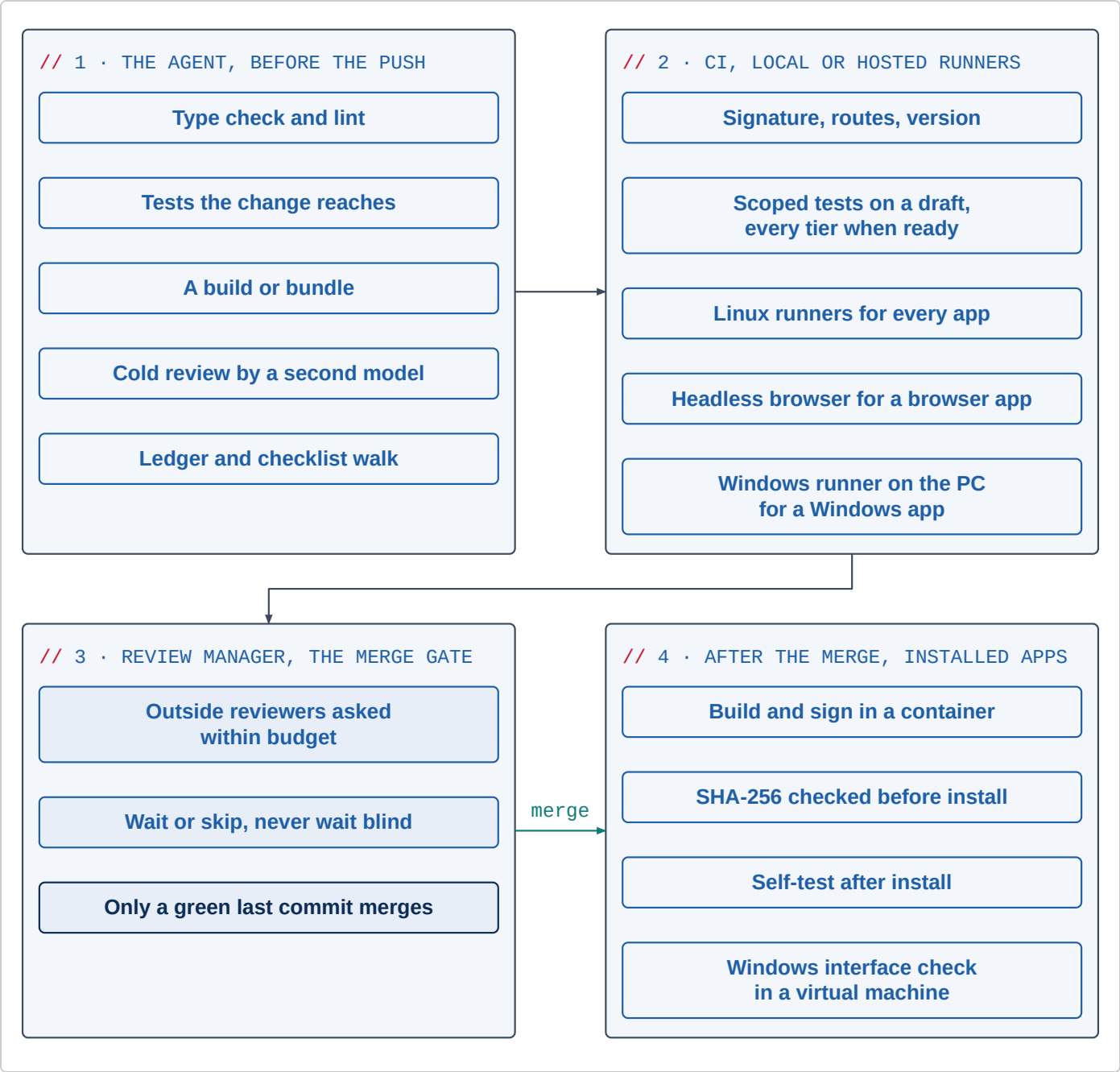


Figure 2. Read the four stages in order: the agent's own checks come first, and only the third stage, the review manager, can let a change merge.

Every new capability gets tests at three levels: unit, integration and system. The lowest level that catches the bug is the one that gets used. A test has to be shown to fail: for a bug, it fails with the defect present and passes with the fix, and the pull request records both runs. Mutation tests, which deliberately break the code to see whether anything notices, exist as a small number of hand-made cases on the files where the stakes are highest. A regression test goes into CI in the same pull request

as the fix for every bug found, and no agent may skip, disable or loosen a test to get a green result.

Tests are selected first and widened when the selection can't be trusted: a script picks them by import analysis and falls back to the whole suite when it's unsure. Recurring defects turn into lint rules generated from ledger entries. The tests are also fenced off from the real machine. After one test signed a real file on the production computer, the operations tests began shadowing the dangerous commands with functions that throw.

Windows applications are tested on Windows only on the production machine. Everything else runs on Linux runners, local or hosted, and a browser application is tested in a headless browser. Mutation testing and security scanning are required by the rules and are run by hand, and the telemetry lists both as not collected.

Stage five: the ledger

Every failure is written down once, as a class, so no agent pays for the same mistake twice. Four ledgers hold 1,151 entries, and 302 of them are classes the desktop application inherited from the frozen web application when it split off.

Table 1. The four ledgers, by repository, with each one's largest groups of failure classes.

REPOSITORY	ENTRIES	LARGEST GROUPS
Desktop application	455: 153 of its own, 302 inherited	interface 34, documentation 28, CI 25
Control service	362: 354 of its own, 8 harvested	runners 121, interface 43, usage 38, builds 25
Web application	325	CI 37, batch scripts 13, git 10, front end 10
Assistant	9 classes, 54 occurrences	startup, installer state, front-end and back-end contract, fixed sizes that crop content

Each entry states when it was discovered and in which pull request, the symptom, the root cause and the prevention. The web application's format adds a search command and an example, and 70 of its 325 entries carry the first and 24 the second. The desktop application assigns a family to each class, such as a check narrower than its rule or a side path around a gate, so related classes are read together. The control service lets a class carry a signature, a fragment of the error line as telemetry shows it, and four classes do.

A failure's life has a fixed order. A test, a CI job, a reviewer, telemetry or the developer finds it. The agent writes a test that fails for that reason. The class goes into the ledger in the same pull request as the fix, or extends an existing entry when it resembles one. Where a check could have caught it, a box is added to the ship checklist and cites the ledger number. Where the pattern is mechanical, it becomes a lint rule or a CI assertion. Before every later push, the agent walks its diff against the ledger. And where the class has a signature, the control service watches for it in telemetry: a class recorded as fixed that appears again raises an alarm, and an error that lasts three days becomes an issue it closes itself when it stops.

Stage six: review

Review happens in two passes. The model that built the change has it reviewed first, cold, before the push. Then the control service runs CI and asks the outside reviewers within each one's budget. None of the reviewers merges, and documentation, diagrams and board cards get no outside review at all.

Cold review has produced the most. In one day, cold reviews of two control-service pull requests that had already passed CI found six real defects, among them a lane that could be double-booked and four ways a real change could skip the test suite. A commercial review app found five more on the same pull requests and a third. That result moved the cold review ahead of the push.

Up to five outside reviewers take the second pass. A local reviewer runs a model on the production machine and reads a code graph for callers and uncovered code, and four hosted services stand beside it: a commercial review app and three reviewers backed by large models. Each has a limit. The local reviewer never runs on a hosted machine,

one model-backed lane stays off documentation-only changes, another is asked only while a budget switch reads on, and the third is paused.

One list decides what is reviewed. A single exclusion file, identical in every repository, generates the path filters and ignore lists the hosted reviewers read, and the CI classifier for documentation-only changes reads the same list. A session never waits blind. Before waiting on any reviewer it asks the control service, which answers per reviewer: arrived, not asked, wait until a stated time, or skip. A reviewer expected within twenty minutes is waited for once. A later one, or one that gives no time, is skipped, and the pull request merges when everything else is green. Each finding gets an answer on its own thread, fixed with the commit or declined with the reason, and a hosted finding is verified before it's accepted, because one reviewer reported a syntax error at the exact point where its own view of the diff was cut off.

Stage seven: continuous integration

One service, the control service, controls every CI machine on the production computer, and CI runs there first. Figure 3 traces the decision for a single job. A Windows job for a Windows application runs only on a Windows runner on that computer. Every other job runs on a Linux runner, either in a virtual machine on the computer or on a hosted machine, and the hosted option is for forks, for jobs that read a secret, or for the weeks the monthly budget allows it. One expression chooses the runner in every workflow, byte for byte the same, so the control service can recognize it. With no switch set, a job waits for the computer and never falls back to a hosted machine by itself.

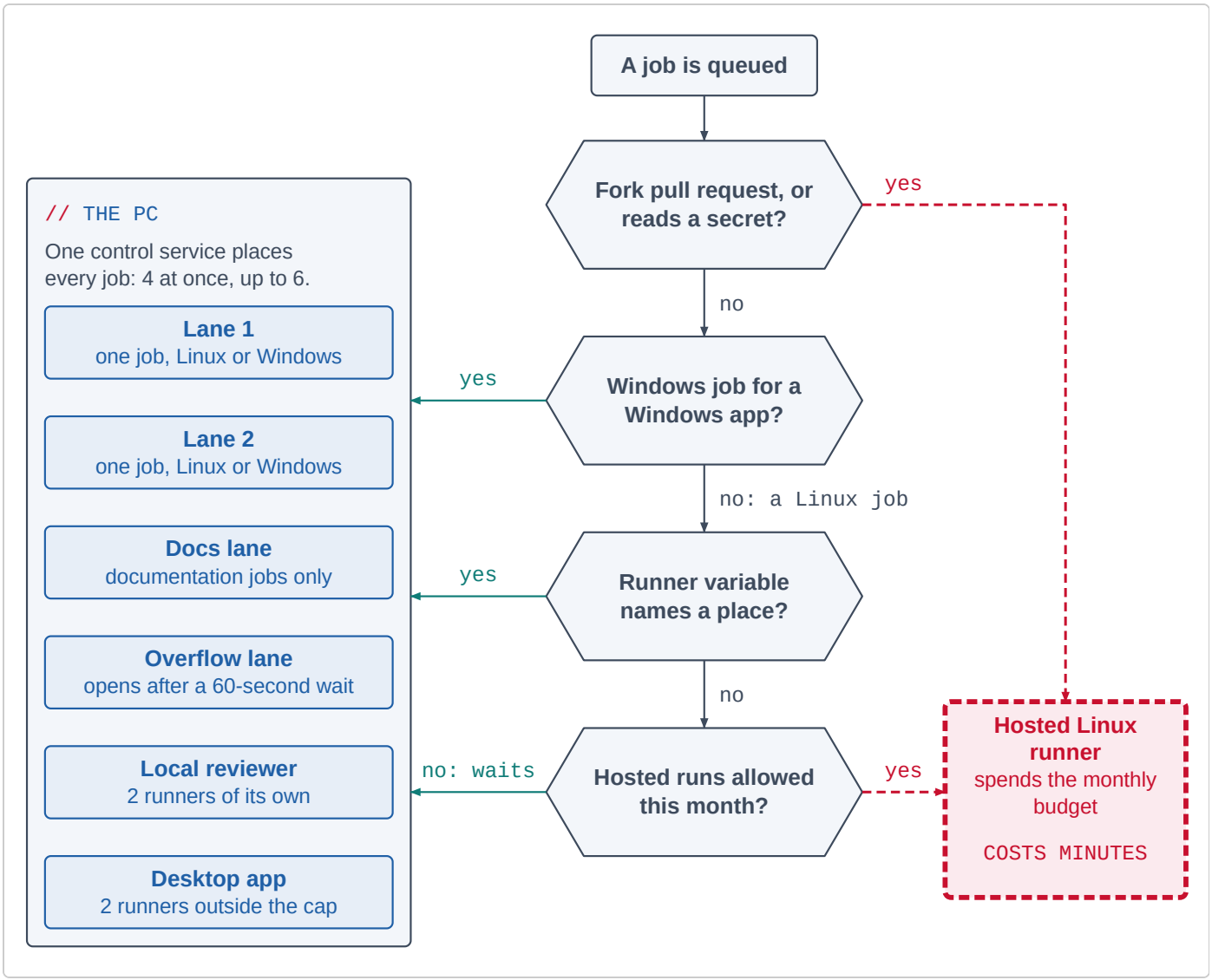


Figure 3. Trace one job down the questions: two answers send it to a hosted runner that spends the budget, and every other answer keeps it on the PC, even when that means waiting.

These rules date from the day the month's 3,000 included hosted minutes ran out, four days into October:

- Local first: every job defaults to the computer.
- The hosted switch is a monthly budget decision, set only by the control service or the developer.
- A job that reads a secret stays hosted, with a comment naming the reason and a sparse schedule.

- Spend nothing on nothing: documentation-only changes take a short pass, drafts skip the expensive jobs, checks run locally before the push, and a round of fixes is pushed once.
- Nobody except the control service starts, stops or renames a runner.
- A pull request from a fork always runs hosted, so a stranger's code never runs on the developer's computer.

Runners are single-use. Each job gets a just-in-time runner that's thrown away afterward, and each repository's Linux runner lives in its own virtual machine cloned from a golden image. Places bound the load: two lanes take one job each, Linux or Windows, a documentation-only runner handles the short pass, and an overflow lane opens once a job has waited sixty seconds. Every placement and every wait is recorded with its reason, so an alarm can name what the plan decided. Actions are pinned by commit hash, workflows declare least-privilege permissions, and a policy script turns the rules into seven checks, including that no job builds on the computer outside four named lanes.

A pull request merges only on a green run of its last commit. One repository makes that a single named check, the aggregate of every test tier and shard. Elsewhere a coding-agent session may approve and merge its own pull request. A pull request is never closed because it's stale or red.

Stage eight: release

A session's part in a release ends at the merge, and Figure 4 follows the path from there. Only an application installed on the production computer gets built. The control service builds the installer in a single-use container with a pinned base image, a quarter of the cores, an 8 GB memory limit and no capabilities, from a read-only archive of the source. It signs the installer with its certificate, a hosted workflow publishes it, and the computer installs it only after checking its SHA-256 hash. A browser application is never built: it runs from the main branch, and a service on the host pulls main every ten minutes and runs any pending one-time script.

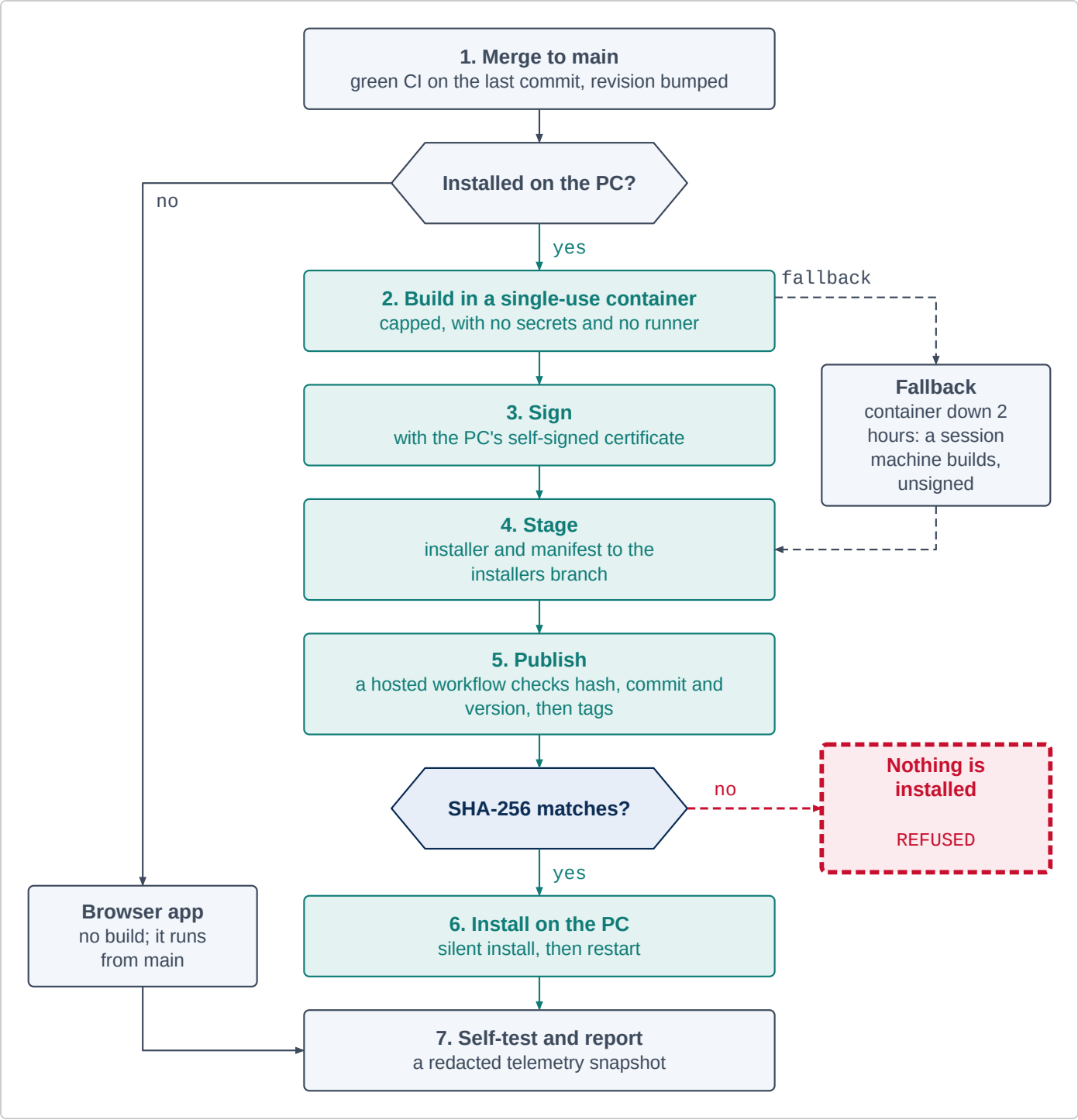


Figure 4. Follow the merge down the center: only an installed app is built and signed, the dashed fallback skips the signature, and a hash that doesn't match installs nothing.

Where builds run changed more than any other rule. They were banned from the computer in October, after one build took every core for ten to fifteen minutes. They moved to the session machine, came back to the computer inside a container three days later when compliance asked for a controlled build environment, and now run in a

container with a fallback: when the container runtime hasn't answered for two hours and a build is due, the control service opens a handover issue and a session builds in its own machine. A build handed to a session machine isn't signed.

Four protections stand around a release. Version discipline: every change other than documentation bumps the revision, and a check compares every file that carries it. Integrity: the manifest records version, commit, hash and the place that built it, and an installer with no hash or a mismatched one installs nothing. Signing: the control service signs the installers it builds, though not yet its own, and the signing certificate is self-signed, so no other machine trusts it. Isolation: a build container has no secret, no network it doesn't need and no access to a runner, and only one of the control service's container builds runs at a time. The dependency audit fails CI on any high or critical advisory, and an advisory with no fix may run under a dated exception that the developer approves, thirty days at most.

Six loops that feed the next session

What production and CI learn goes back to the next session through six loops. Telemetry comes first: the control service writes a redacted snapshot to an orphan branch about every ten minutes, a scheduled task covers for it while it's closed, and the start-of-session hook prints the digest and marks it fresh or stale. Error follow-through comes second. Since the ledger classes learned to carry a signature, a fixed class seen again raises an alarm and an error seen three days running becomes an issue.

History and bottlenecks form the third loop: CI ledgers by month, billing by repository, token counts, and 44 bottleneck questions each answered as measured, could not look or not collected. The fourth is CI learning, a job that reads each shard's test results every ten minutes and opens a card for any test that passes and fails on one commit twice, closing it after fourteen clean days. Reviewer learning is the fifth: the local reviewer's findings are adjudicated from git and from replies on the thread, so a finding the next commit fixed becomes confirmed and a reply can mark it refuted. Git alone never writes refuted, because it can't tell wrong from ignored. The sixth is the exchange between repositories: shared rules are byte-identical and synced by pull

request, failure patterns are harvested rather than copied, and a weekly check looks for drift.

Session end closes the loop. Before a session ends it records what shipped and what's unfinished on the board, updates the handoff with the branch, the revision, the test results, the pull request's status and the exact next step, adds any new failure class to the ledger and any new rule to the standing-rules file, and pushes. A hook refuses to end a session that holds uncommitted work. Two rules govern all six loops. Absent is not zero — a probe that couldn't run reports could not look and never an empty list. And telemetry carries metrics and error text through a redaction step, never prompts, replies or file contents.

Where it broke

One dominant pattern runs through the ledgers. Moving CI onto the developer's computer fixed the hosting bill and created the runner, memory, container and build problems that fill most of the entries since. A second pattern runs under it: a rule written in prose kept failing until it became a check, and each new check then needed one to three fixes of its own. That's how the process got its shape. Table 2 gives each problem a verdict from the record, where worked means no recurrence is logged, partly means later entries of the same kind exist and failed means the fix didn't hold or was never applied.

Table 2. Fourteen problems from the ledgers, with the size of each and its verdict today.

PROBLEM	SIZE OF IT	VERDICT
Self-hosted runners stall, queue or drop jobs	121 entries in one ledger plus about 12 in two others; waits of nine, ten and thirteen hours	Partly
Hosted minutes ran out	3,000 of 3,000 minutes gone four days into October	Worked for cost, and caused four of the other problems
The container runtime and the Linux virtual machine break builds and CI	the computer sat twelve versions behind main	Partly
Where builds run kept changing	five rulings in eight days; main failed to compile for a spell	Partly
Merges over red, running or absent checks	at least nine pull requests across three repositories	Failed
Outside reviewers ran out of quota or were withdrawn	five services lost or paused between July and November	Partly
The code host's hourly API limit exhausted	four times; up to 25,000 calls an hour against a limit of 5,000	Partly
Conflicts on generated shared files	a dozen rebases in one day	Worked for the board
Scoped, draft or documentation-only CI passed broken code	three pull requests broke main in one day	Partly
Sessions and subagents skipped the routine	six finished slices on no remote branch	Partly
The local reviewer was unreliable and didn't yet learn	seven red runs in a row on one pull request	Partly

PROBLEM	SIZE OF IT	VERDICT
A fix was recorded and never checked	one error grew a hundred sixty-fold after it was marked fixed	Worked on its first case
A gate with no fix to apply	an advisory with no patched version stopped every pull request	Worked for now
Stale telemetry, handoffs and documents	a day and two releases spent on an outage that didn't exist	Worked for telemetry

Runner trouble deserves its own account, because it filled the most entries and its fixes show the pattern most clearly. A hold process to keep the runners alive failed and recurred twice. Fixed lanes gave way to places assigned from the waiting jobs, and more than thirty entries followed in four days. A keeper for automatic repair came next, then a cap of four jobs, then single-use runners, then a health watch that kicked a stuck runner. That watch never counted the kicks it refused, so telemetry said pass while every Windows runner was dead. The turn came when the last ten allocation classes were sorted into five families: rules kept in more than one place, a stale read of the code host, state lost on a restart, operating-system and job matching, and a health check that disagreed with the planner. The fix aimed at the families and not at the next instance: one definition of a place's state, the rules in one table, invariant tests over generated queues, a restart test, a stale-read guard and a record of every decision. A cold review before the merge found a lane that could be double-booked and a guard that would have spent about 190 API calls an hour, and both were fixed. The proof is still pending — the test is seven days of telemetry with no allocation alarm that the decision record can't explain.

A smaller story, the hosted budget, has a plain moral. Fourteen full test pyramids ran on three draft pull requests in three hours, and one review lane alone cost about 1,200 minutes a month, two-fifths of the plan's 3,000. The fixes ran in order: CI limited to main pushes, the expensive lanes only at open and at ready, then local-first rules. Table 3 shows the bill falling, and then the cost moved to the computer.

Table 3. Hosted CI minutes and the amount billed, by month, from the billing history.

MONTH	HOSTED MINUTES	BILLED
June	38,707	$214.19
July	38,420	$189.17
August	35,073	$230.48, a review subscription included
September	31,765	$165.86
October	8,522	$32.27
November, to the 18th	2,028	$0.00

November isn't finished; by the 18th, 2,155 of the 3,000 included minutes were counted against the allowance, which at that pace runs past the allowance late in the month.

The container runtime and the Linux virtual machine broke builds in a way that's still unexplained. The container runtime restarted twenty-six times in three hours with no release, and the production computer fell twelve versions behind main. Restart backoff, a full revive, a different virtualization backend and a stall killer all helped partly. The backend switch broke the Windows interface checks, so containers inside the Linux virtual machine came back for those checks only. Two ninety-minute hangs followed, and the ledger says plainly that why the runtime hung is not known.

Merging over red is the problem the title names. Two control-service pull requests merged red and broke main for everyone after them. One merged three seconds after a reviewer posted nine findings — nobody had read them. Another merged forty-four seconds after its shards started, two more before CI ran on their last commit, and a third after a five-hour wait with no CI at all. Two more merged on the developer's own word over a red audit step and before the run on the last commit had finished. The rules reduced it, though never to zero. The one mechanical guard is a required status check on the main branch, which is a setting on the code host that only the developer can make, and it isn't set in two of the repositories. The code host reports one of them as not protected. The verdict stays failed until the setting exists.

One failure is worth its own paragraph because the process caught itself on it. A page error appeared at eleven lines a day, was fixed, was entered in the ledger and was given a test in CI. It then grew to 1,759 lines a day. The test looked for the text of a constructor call, and the panel tracker built its observer through a variable. Nothing compared a class marked fixed with what telemetry still reported. The developer had both pieces built in one day: the error follow-through that watches a fixed class's signature, and the real fix, which reproduced the error in a browser, found a panel resizing itself by a fraction of a pixel on every frame, and stopped it. The watch raised its alarm that same day, and the next snapshot showed no fixed class seen again.

What's still to build

The change with the widest effect costs nothing to build: make the CI check a required status check on main in each repository. It's a setting that belongs to the developer, and it turns the merge rule from text into a gate. A few more decisions sit in the same place: a signing certificate other machines would trust, who sets the version (the pull request or the merge), and what to do about a dated exception that expires within the month.

The build list runs longer and plainer. Windows runners through the runner broker, with autoscale and a clean step before each run. A compile gate that doesn't load the computer. A release workflow for the assistant, which has none. The control service signing its own installer the way it signs the others. A front-end test runner where none exists. Automated mutation testing and security scanning, which the rules require and the hands still do. Eight bottleneck metrics that read not collected, among them subscription cost, cloud tokens, time spent waiting on the developer, and build and interface-check times. And a channel between sessions that doesn't pass through the developer.

Rule files carry the last open item. Every rule file still holds a process the developer retired in November, kept as history, and every session loads that file as context. One of them runs to 222,034 bytes, and at its last measurement it had used 44,784 of its 55,000-token budget, about four-fifths. The record found 29 places where a rule file and the code, or two rule files, disagree: a handoff that calls a pull request open after it merged, a schedule that three documents give three ways, a rule number used three

times, a timeout that reads eighty minutes in a comment and ninety in the code. Each one misleads the next session that reads it.

Why it matters

A process run by agents shows exactly which of its rules are machines and which are sentences, because an agent follows a sentence only as long as nothing pulls against it. The sentence about merging on green held on most days and broke at least nine times, in the hands of agents and of the developer, each time under the pressure of a long queue. The rules that never broke were the ones with a checker behind them, and the checker was often the fix for an earlier break in the same rule.

The same record argues for building the checker at the first failure and not the fifth. A ledger class costs an hour to write and a family costs a day to find, while an unenforced rule costs a release, and sometimes two. A team that adopts agents for the build will find its own version of that status check: a rule everyone agrees on, written in a file, enforced by memory. The question to ask of any process is which of its rules would still hold if every person and every agent stopped caring tomorrow, and the answer is the list of what's actually been built.

Sources

Every count, date and dollar figure here comes from the developer's own review of the four repositories at their main heads on 18 and 19 November 2026, from the control service's telemetry snapshots of 17 and 18 November, and from its billing history of 18 November. Dates and outcomes in the account of what broke come from the ledgers and session logs. Tables 1, 2 and 3 are the complete data from that review and take no appendix.

Glossary

TERM	DEFINITION
Agent	An AI model given tools and a task, such as writing a change on its own branch.
Board	The list of requested items, written before any code so a request outlives the session that took it.
Code host	The service that holds the repositories, runs the hosted CI machines and enforces branch settings.
Cold review	A review by a model that sees only the diff, with no access to the builder's reasoning.
Container	A sealed, throwaway environment for a build, with its own limits on cores, memory and network.
Continuous integration (CI)	Automatic builds and tests that run on every change before it merges.
Control service	The program on the production computer that places CI jobs, asks reviewers, builds and signs installers and collects telemetry.
Dated exception	A recorded, approved waiver of a failing check that ends by itself on a fixed day.
Handoff	The file where a session leaves its branch, results and exact next step for the session after it.
Ledger	A file of failure classes, each with its symptom, root cause and prevention, read before every push.
Merge gate	The condition a pull request must meet before it merges, here a green run of its last commit.
Mutation testing	Deliberately breaking code to confirm that a test notices.

TERM	DEFINITION
Pull request	A proposed change to a repository, reviewed and tested before it joins the main branch.
Required status check	A branch setting on the code host that blocks a merge until a named check passes.
Runner	A machine that runs a CI job.
SHA-256	A hash function whose output changes when any byte of a file changes, used to check an installer.
Single-use runner	A runner created for one job and discarded afterward.
Telemetry	Metrics and error text sent from a running system to its maintainers.

CLOUD / DEVOPS ART-CLD-013

// 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 →