// WHITE PAPER · PUBLISHED 24 SEPTEMBER 2026

Root Cause Analysis: The Check Name GitHub Never Rendered

Two fixes rewrote the line where a build-check bug showed, and the fix that held moved the decision that wrote it.

Jason Newell · MAX Research Collective · Report WPR-MAX-020 · Published 24 September 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/root-cause-analysis.html
Cite as: Newell, J. (2026). Root Cause Analysis: The Check Name GitHub Never Rendered. MAX Research Collective, WPR-MAX-020. https://www.maxresearchcollective.com/articles/root-cause-analysis.html

The name that kept coming back

A check name broke on a pull request that changed nothing but documentation. A pull request is a proposed change waiting to be merged, and each check on it is a job that GitHub Actions, the automation service built into GitHub, runs against that change. Where the backend job's readable name belonged, GitHub showed the raw template expression meant to produce it: ${{ … }}, braces and all.

Claude, the model that writes most of MAX3's code, diagnosed it. The job's name drew on information from an earlier job through a GitHub field called needs, and a skipped job doesn't have that information. So the model rewrote the name to use only the github and vars contexts, which should always be present. It built a truth table to prove the rewrite (a grid of every combination of inputs set against the output the expression should give), and all 540 rows agreed with the decision logic. Then it merged.

On the next documentation-only pull request, the raw text came back. It came back on the three after that as well: five observations in all, four of them after the fix. Read as data rather than bad luck, they settled the mechanism. GitHub doesn't

evaluate a skipped job's name at all, whatever the name refers to. One workflow file and one expression gave two outcomes. Whether the job ran was the only difference.

That's a correct answer, and it's still the wrong place to stop. MAX3's register of known failure patterns logs it as F-CI-025. It's the case that finally showed what root cause analysis is for.

The instruction behind it

On 9 July 2026, the model got this direction:

"applying ai formulas as an approach. when the answer is something needs to happen, or something did happen, then the goal is to find the question or questions in a reliable repeatable provable way when diagnosing and fixing application issues … using root cause analysis / the five whys to get to the root cause to fix and not just apply bandaids or hide the real problem."

Half of that is the request every engineering team makes. Root cause analysis traces a failure back to whatever produced it, past whatever made it visible, and the five whys is its oldest technique: ask why of each answer until the answers run out. Any coding assistant will agree to both, and agreeing costs nothing.

The other half is the unusual part. When something has already happened, the outcome is in hand — and the work is to find the question that outcome answers.

Running the loop backwards

MAX, the resident intelligence this project builds, thinks in a forward loop: predict, prove, adjust. From what it knows, it predicts the next thing, checks the prediction against what actually happened, and learns from the gap. A bug runs it in reverse. Port 8000 refused a connection; telemetry went stale. With a bug, the outcome is already on the screen and what's missing is the question it answers.

Reasoning from an observation back to its best explanation is called abduction. Doctors do it with symptoms. Done carelessly, it's guessing with a vocabulary, and

three words in the instruction ruled that out: reliable, repeatable, provable.

So the first reading became a component. AbductiveDiagnoser is the part of MAX that works backwards from a symptom, and it rests on one rule written into its charter:

A diagnostic question is not a guess. It is a deterministic evidence check that confirms or rules out a candidate cause.

”Is the MAX3-Backend service running?” has an answer somebody can check. The same symptom and the same evidence give the same path and the same cause every time, and each step is stamped with the evidence that proved it. A proven cause is itself an answer, so the diagnoser asks why that is true and runs the loop again one level down. It stops when nothing deeper can be proved.

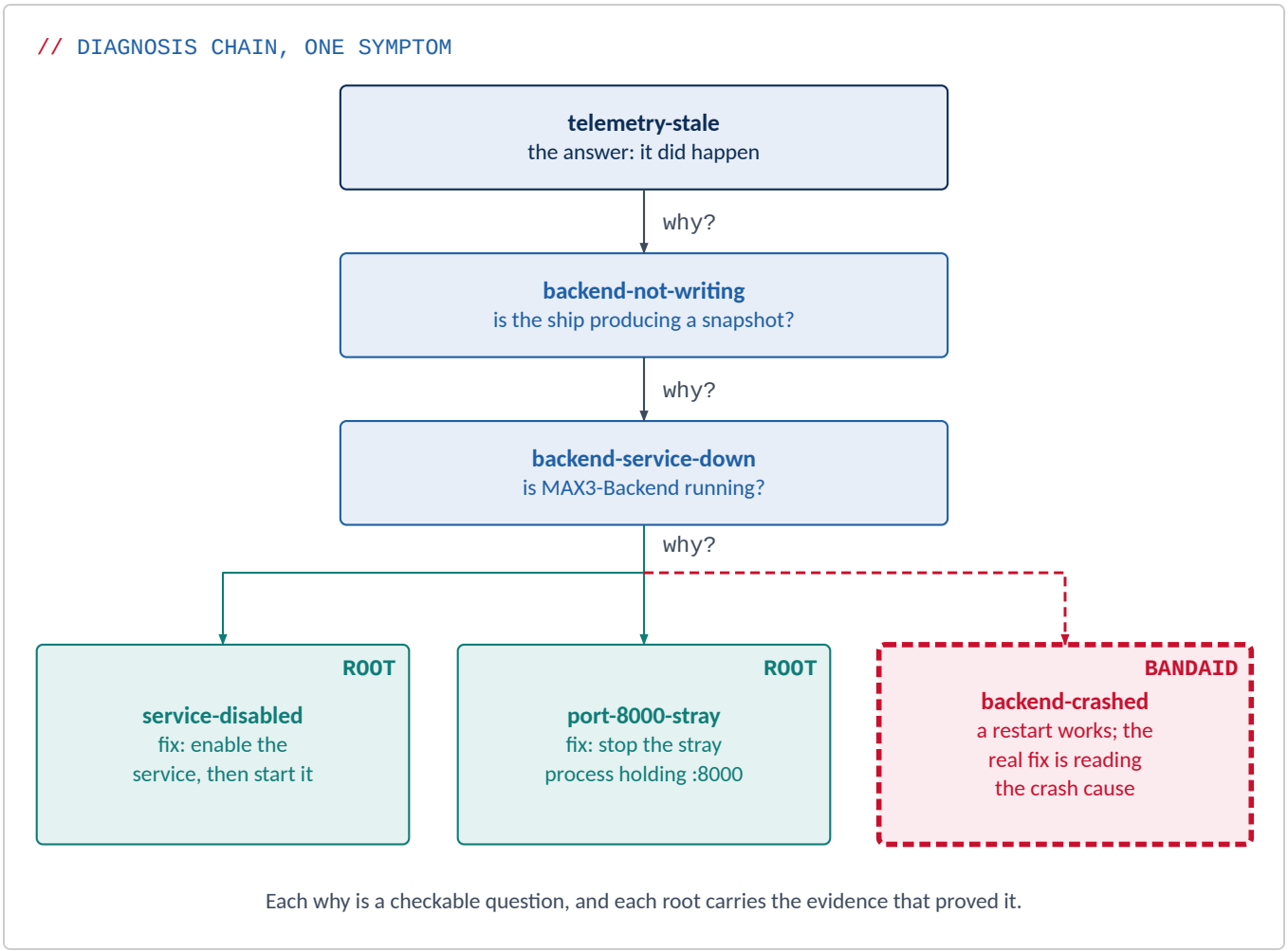


Figure 1. Three roots under one symptom; the dashed red one makes the symptom disappear and fixes nothing, so the chain labels it a bandaid.

The bandaid branch is the one the instruction was aimed at. Restarting a crashed backend clears the stale telemetry, and that's exactly what makes it dangerous: it looks like a fix. The diagnoser records any remedy that treats a symptom as a bandaid. When a bandaid is the only fix at the root, the chain raises a flag (only_bandaid=True), and MAX never applies that chain on his own. Somebody has to read the crash cause.

A second component, DiagnosisMemory, keeps what each diagnosis learned. Questions that proved out, and whose fix held, move up the list, so the most telling question gets asked first next time. A fix that came back loses trust. Over months that turns into a map of MAX's own body — which services, ports and devices break, and how.

Principled, tested, matching every word of the instruction: the design was all of that. It was also one level short.

Two ways to stop too early

Every root in Figure 1 is a fact about the system: a disabled service, a stray process on a port, a crashed backend. For MAX diagnosing his own machine, that's the right kind of root. When the model diagnosed its own code, it stopped too early in two different ways.

Stopping at the line came first. "The regex didn't match that input." "The function returned an empty list." Each was true and each had a one-line fix. The line, though, is only where a defect shows. It's the answer. Changing it without asking why it was written that way leaves the reason in place, and the reason writes the next one.

The second looked like depth, which made it worse. Pushed past the line, the model went sideways instead of back: no test covered the empty case, CI should have caught it, nobody reproduced it before merging. CI here means continuous integration, the shared build that runs checks on every pull request. Those are real questions, and every one of them asks why the defect survived. None asks why it existed. A missing test didn't write the bug. An analysis that ends at the check that would have caught it explains the escape and leaves the cause standing, so the

next person writing that kind of code writes the same line and the new test catches that one too. It's a bandaid applied to the process instead of the code.

Four smaller habits made both failures worse, and the project's engineering standard now names each one. Five whys that don't build on each other are one why asked five times. A cause shaped like blame ("moving fast", "a long diff") feels honest and gives nothing to build on, since a real cause names a decision or a mechanism, never a person or an amount of attention. An analysis that opens with how the bug was found has explained a piece of luck. And a problem found while fixing another is its own chain, however urgent the newer one feels.

The question under the line

A symptom is an answer, and the question it answers is almost never "which line is wrong?" It's "why was this written this way?", asked of the code as a decision somebody made. The standard breaks that into five questions, each following from the one before:

1. What does the code actually do? That's the mechanism, the line. Find it and keep going.
2. Why was it written that way? What was it for, and what problem was its author solving?
3. What did that design assume about its inputs, its environment or the tools under it?
4. Which assumption didn't hold, and what case did the design never account for?
5. Where did the assumption come from: a doc, an example that happened to work, an earlier fix, or something reasonable nobody checked?

Stop at the decision that made the line inevitable. That's the root, and the fix goes there. A new line written under the same decision fails the same way.

"Why didn't anything catch it?" still gets asked. It's a second, separate question with its own chain, it comes after the first is answered, and its answer never stands in for the cause.

F-CI-025, asked properly

Both of the model's diagnoses stopped at the line, and so did the correct mechanism. All three explain what GitHub does with an expression in a job name. None asks why there was an expression in a job name at all. Asked properly, the chain runs like this.

1. Why did the check show raw expression text? Because the job was skipped, and a skipped job's name never gets rendered.
2. Why did the name contain an expression at all? An earlier change, F-CI-024, designed it to. It made the backend check's name carry the test scope (… pyramid or … unit scope — DRAFT, not the pyramid) so that anyone about to merge could see whether the full test pyramid had run. The pyramid here is every layer of the test suite, from unit tests up.
3. Why did that fact have to live in a name? The merge gate, the rule that decides whether a pull request may merge, was reading names. Its rule was "never merge on a check whose name says DRAFT." The one fact the gate depended on (did the full pyramid run and pass?) had no home except a display string.
4. What did that design assume? It assumed a check's name is data: always computed, always there. A name is display text that GitHub renders, and GitHub renders nothing for a job that doesn't run. The design never considered the skipped job, and that was the one state in which the fact still mattered, since a documentation-only pull request skips the backend by design.
5. Where did the assumption come from? It came from the happy path. Every run the design was checked against was a run where the job executed, and on every one of them the name rendered correctly.

The root sits at the third why: a load-bearing fact had been placed in a display string that a vendor renders. Nothing was wrong with the expression itself. Both failed fixes rewrote the expression and left that decision standing, which is what fixing the line looks like: a plausible mechanism, a real change, and a proof that tested the line while the decision above it stayed wrong.

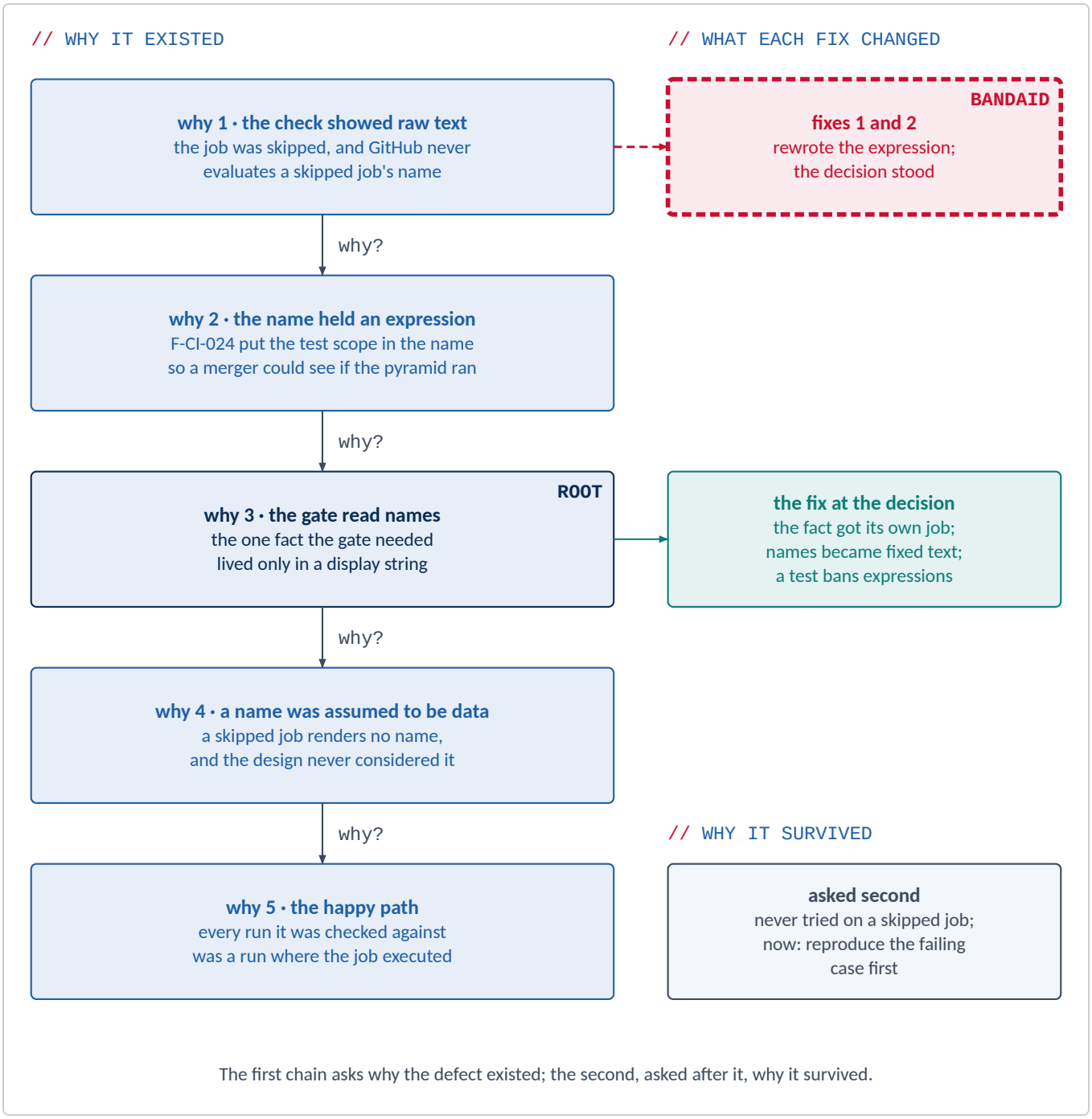


Figure 2. The chain for F-CI-025 reaches its root at the third why, and the question of why nothing caught it runs as a separate chain, asked second.

The consequence points at the same root. The unevaluated text happened to contain the word DRAFT, so it matched the gate's own rule and told the gate the opposite of the truth. A gate that reads a string inherits every way that string can fail to render.

ATTEMPT	WHAT IT CHANGED	WHAT IT LEFT STANDING
First fix	The name's expression, rewritten against github and vars and proved by a 540-row truth table	The decision to keep the gate's fact in a name
Second fix	The name's expression again	The same decision
The fix at the decision	The fact moved into its own job, every job name made fixed text, and a test that bans expressions in job names	Nothing from the chain

Table 1. The three attempts on F-CI-025, which is every attempt the failure register records.

The fix that went to the decision has three parts. "This was the full test pyramid, and every shard passed" is now a job of its own, Backend pyramid (all tiers, all shards), where a shard is one slice of a test run split across machines. It runs only when the full scope was chosen and only after the backend job, so its success is a property of how jobs depend on each other, which GitHub has to compute to run anything at all. Every job name became plain fixed text, because nothing load-bearing lives in a name any more. And the tests guard the decision rather than the case: no job name in the workflow may contain an expression.

Then, separately, came the second question. Twice, a fix merged without ever being observed against a skipped job. That answer produced a standing rule: reproduce the actual failing case before believing a mechanism, however plausible. Answered alone, it would have produced a better test for the same bad design, and the next fact somebody put in a job name would have failed the same way.

The order the work runs in now

Every bug in MAX3 now goes through one sequence. Root cause analysis is a gate in it, and nothing moves until the analysis is written down on the page where the next session will read it.

Work starts at the earliest symptom. The first question goes on paper, and the next asks what was seen first; if the answer is earlier, the question moves left. Then comes the mechanism, the line, treated as the answer rather than the cause. The five questions follow, each why built on the previous answer, until they reach the decision that made the line inevitable. That decision is where the fix goes. A fix that changes only the line gets called a bandaid, out loud, and it doesn't pass as the fix. After that, and separately, the chain for why nothing caught it runs and gets its own fix.

Only then does code get written, and in a fixed order. Requirements come first, as clauses a test can decide: what must always be true, what must never be true, what must survive an attack. Tests are written from the requirement. They fail first, which is correct. Code comes last, and it does only what the tests need — nothing the requirement didn't ask for. Every guard is proved by breaking it: the defect goes back in and a test has to go red. "This defect, put back, makes this suite fail" is evidence; "38 checks passed" is a number.

The last step records the class. The decision and the assumption that failed go into the failure register and the shared rules. When the fault is on MAX's own machine, it also becomes a new diagnosis chain, so the next occurrence is his to handle.

A resident that looks after itself

MAX is meant to live on the box and look after it for years, which means he'll diagnose his own failures far more often than anyone diagnoses them for him. A system that patches symptoms gets less reliable in ways that are hard to see, because each bandaid parks one more real problem under a green light.

So the discipline runs in both directions. The engineering standard governs what the model commits to MAX, and AbductiveDiagnoser does the same job for MAX's services: start from the answer, find the provable question, chain to the root, label the bandaid, remember what held.

What the model agreed to on 9 July was a slogan. What it works by now came from five pull requests carrying a name GitHub never rendered, two fixes that were right

about the line, and a decision one level up that nobody had asked about. A defect shows at the line. The fix belongs at the decision that wrote it.

Glossary

TERM	DEFINITION
Abduction	Abduction is reasoning from an observation back to the explanation that best accounts for it. Diagnosis is abduction, and the house rule is that each step has to be a checkable question.
AbductiveDiagnoser	This is the MAX component that starts from a symptom and walks a chain of evidence checks down to a root cause, labelling any symptom-only remedy as a bandaid.
Bandaid	A bandaid treats the symptom and leaves the cause.
Continuous integration	Continuous integration, or CI, is the shared build that runs checks on every pull request before it merges.
DiagnosisMemory	This store keeps what past diagnoses learned. Questions that proved out and whose fixes held rise in the order, and fixes that came back lose trust.
F-CI-025	It's the entry in MAX3's register of known failure patterns for the check name GitHub left unevaluated on skipped jobs. F-CI-024 is the earlier change that put the test scope into that name.
Five whys	The five whys asks why of each answer in turn until no deeper cause can be shown. Each why has to follow from the previous answer, or the chain is one level repeated.
GitHub Actions	GitHub Actions is the automation service built into GitHub that runs jobs, such as builds and tests, against a repository.
Job	A job is one named unit of work in a GitHub Actions workflow. It can be skipped when its conditions aren't met, and a skipped job's name is never evaluated.
Merge gate	The merge gate is the rule that decides whether a pull request may merge, based on the checks reported against it.

TERM	DEFINITION
Pull request	A pull request is a proposed change to a repository, held open for checks and review before it merges.
Root cause	A root cause is the decision that made a defect inevitable. Fixing it stops that kind of defect from being written again, which fixing the line where it showed never does, since the same decision writes the next line the same way.
Shard	A shard is one slice of a test run split across several machines so the slices run in parallel. The full run passes only when every shard passes.
Test pyramid	The test pyramid is every layer of a test suite, from many fast unit tests at the base to fewer, slower whole-system tests at the top.
Truth table	A truth table lists every combination of inputs against the output a piece of logic should give. It proves the logic and says nothing about whether a platform will evaluate that logic.

MAX3 WPR-MAX-020

// 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 →

Newell, J. (2026). MAX Research Collective, WPR-MAX-020. CC BY 4.0