The Epistemology of Repair

Flyxion
Independent Researcher

2026

Abstract

A working system withholds most of its own internal structure, because too many possible internal arrangements are compatible with the single fact of currently functioning correctly. Failure changes this. This essay argues that breakage is not merely the loss of function but an information-producing event, and that repair is best understood as a repeatable procedure of inference conducted under severe information constraints: observe a failure, infer a narrowed space of hypotheses about inaccessible state, choose an intervention selected for what it will distinguish among those hypotheses rather than for how completely it restores function, and read the system's subsequent behavior as evidence. The argument is developed across three registers within software specifically, dependency resolution, memory corruption, and race conditions, chosen because their internal dynamics are largely invisible even to people who use the software daily. From there the essay develops a distinction between the restorative and discriminatory value of an intervention, treats the ordinary workaround as an underappreciated form of causal knowledge, and considers what determines whether diagnostic information, once produced, survives to be useful again.

1 The Problem with Working Systems

A system that has been running correctly for a long time provides surprisingly little evidence about its own internal structure. Sustained correct operation does rule some arrangements out, particularly once the system has encountered a wide range of inputs, but it remains comparatively weak at discriminating among whatever arrangements survive that filtering. Uptime is compatible with a surprisingly large class of underlying structures, so long as each one happens to produce the right external behavior on whatever the system has actually been asked to do so far. A dependency graph that resolves cleanly might be doing so because its versions are genuinely compatible, or because a conflict exists but has never been triggered by any input the system has processed yet, or because two separate

faults are canceling each other's effects. All three possibilities look identical from outside. Correct behavior does not distinguish between them.

This is easy to overlook because working systems are the ones people spend the least time examining. Attention follows failure, not function, so the internal organization of anything currently working tends to stay unexamined by default, not because it is simple but because nothing has forced anyone to look. The knowledge required to build a system and the knowledge required to explain why a particular working instance of it behaves the way it does are not the same knowledge, and the gap between them can persist indefinitely as long as nothing breaks.

2 Breakage as an Information-Producing Event

Failure changes this. The moment a system stops behaving as expected, the space of arrangements compatible with its behavior contracts, often sharply. A dependency conflict that could have been anything now has to be something specific enough to produce this particular error, at this particular point, under these particular conditions. The failure has not damaged the system's structure, in most cases, so much as revealed a constraint that was already there and simply had no occasion to matter before.

This suggests a claim stronger than the common observation that failures are useful for debugging. Breakage is not incidental to understanding a system; it is frequently the primary mechanism by which understanding becomes possible at all, for anyone other than the system's original designer. The designer's model reflects intent, how the system was meant to work. The model available to whoever encounters the system later, and has to contend with what it does when it stops working, reflects something different: the actual space of ways the system can fail, learned one incident at a time. These two models can diverge substantially, and when they do, it is frequently the second one that turns out to be more accurate about how the system behaves under stress, precisely because it was built out of failures rather than intentions.

This also implies something about the relative evidential weight of success and failure, worth stating plainly rather than leaving implicit. A long run of correct executions, across a wide variety of inputs, does narrow the space of arrangements compatible with the system's behavior, but it narrows slowly and unevenly, since a great many internal structures remain equally capable of producing correct output on whatever inputs have actually been tried. A single well-characterized failure narrows that space far more sharply, because it has to be compatible not just with everything the system got right but with the specific way it went wrong. Success accumulates weak evidence gradually. Failure, handled well, produces strong evidence all at once. Neither is preferable in a practical sense, a system that never fails is obviously better to depend on than one that fails often and gets diagnosed for it, but the two kinds of evidence are not symmetric, and treating them as though they were is a common source of overconfidence in systems that simply have not yet been tested hard enough to fail.

There is also a difference worth naming between repair and laboratory science, even though the underlying inferential structure, hypothesis, intervention, observation, is the same in both. A scientist designs an experiment specifically to discriminate between live hypotheses. A repairer inherits an experiment that reality has already run without being asked: the corrupted heap, the failed build, the race condition that appears under load. The craft of repair lies less in deciding what experiment to run, since the experiment has already happened, than in reconstructing enough of its conditions to interpret what it showed, and then designing a second, deliberate experiment, the intervention, to follow up on it. Repair is experimental inference performed on phenomena the repairer did not get to design.

This affinity with laboratory science is real, but the resemblance to any tidy image of scientific method is weaker than it looks. Paul Feyerabend argued, drawing on the history of astronomy and mechanics, that scientific progress has owed at least as much to methodological opportunism, reaching for whatever approach makes progress in a given case and discarding standard procedure when it fails to, as to consistent adherence to a single method. Repair fits that description more comfortably than it fits any formal protocol. A repairer does not apply one procedure uniformly across cases; a dependency bisection, a memory sanitizer, and a deliberately controlled reproduction of a race are three unrelated tools, reached for because each happens to discriminate in its own domain, not because they descend from a shared method. If repair counts as science, it is science of the opportunistic kind Feyerabend described, assembled case by case, rather than the kind found in a methods textbook.

What follows treats this as a procedure rather than a single observation: a repeatable structure of inference that failure makes available and that has to be actively used, through deliberately chosen intervention, or the information a failure produced simply goes unclaimed.

3 The Shape of Diagnosis

A build that worked yesterday fails today after a routine dependency update, and the visible symptom, a type mismatch, a missing export, an undefined function, rarely points at its own cause. Modern dependency graphs run many layers deep, so the package that actually changed is often several transitive steps removed from the package whose error message appears on screen. The naive response is to roll the whole lockfile back to the last known-good state, which restores function but teaches nothing, because dozens of package versions moved at once and no single one can be credited with the fix. The more disciplined response pins one dependency at a time, rebuilding after each change. This does not isolate a single responsible package as cleanly as it might appear to. Some failures arise not from any one package's version but from an interaction between two packages' versions, so changing one while holding the other fixed can leave that interaction hypothesis completely untested, and a build that still fails after pinning package A says nothing about whether A was ever the problem in isolation. What the discipline actually isolates, strictly, is not a component but a hypothesis: the claim under test at that step is not "package A is at fault" but something narrower, closer to "reverting A alone, with everything else held constant,

restores the property." A negative result rules out exactly that claim and nothing more.

A different failure mode makes the same structure visible from the inside of a running process rather than from outside a build system. A program corrupts memory, writes past the end of a buffer or frees something still in use, but the crash typically does not happen at the site of the corruption. It happens later, whenever some other part of the program finally reads the damaged memory and produces behavior severe enough to halt execution. The stack trace at the moment of the crash marks where the bad state became intolerable, not where it was introduced, and those two locations can be separated by an arbitrary number of intervening operations. Tools built specifically for this problem, memory checkers, sanitizers, reduced test cases, exist because narrowing the gap between corruption and detection is a distinct technical problem from fixing the corruption itself once it is found.

Race conditions push the same structure to its sharpest form, because in a race condition the act of observing the system can change its behavior. A bug that reproduces reliably in production can vanish the instant a debugger attaches or a print statement gets added, because either one alters the relative timing of the threads involved, and timing is the entire mechanism of the bug. This turns intervention and observation, elsewhere separable in principle even if separating them takes discipline, into something that can be causally indistinguishable in practice: the tool used to look is frequently a perturbation to the thing being looked at. Diagnosing this class of failure usually requires giving up on direct observation altogether and instead reasoning about invariants, what states the threads are permitted to be in relative to each other, and which of those permitted states, if reached, would produce the observed symptom.

Three registers, three completely different failure mechanisms, and the same underlying procedure running in each: a symptom that under-determines its cause, a hypothesis space narrower than "anything is possible" but wider than "the cause is known," an intervention chosen for what it will distinguish among the surviving hypotheses rather than for how quickly it restores function, and a continuation read as evidence that narrows the space further. What a discriminating intervention isolates, in every register, is a hypothesis rather than a component of the system, and the two coincide only when the underlying fault happens not to involve an interaction between parts. When it does involve one, as it often does, a clean negative result about one component can still leave an entire class of interaction hypotheses completely untouched. What differs across the three cases is only how expensive good diagnostic intervention is relative to careless restoration, and in the race-condition case, whether direct intervention is available as an option at all.

4 Restoration and Diagnosis as Separate Objectives

Every intervention available to a repairer can be scored two different ways. One score asks whether the intervention restores the desired behavior. The other asks how much it narrows the space of competing explanations for why the behavior was lost in the first place. These are not the same measurement taken twice. An intervention can rank highly on one

and poorly on the other, and the gap between them is where most of what looks like diagnostic skill actually lives.

The dependency case makes the gap concrete. Rolling a lockfile back to its last known-good state has close to maximal restorative value: the build passes, the symptom is gone, the immediate problem is solved. Its discriminatory value is close to zero, because dozens of packages moved simultaneously and the rollback cannot say which of them mattered. Pinning one dependency at a time has lower restorative value in the short run, since the build may still be broken after the first several attempts, but each failed attempt eliminates one candidate cleanly. The full rollback and the incremental pin converge on the same eventual state, a working build, but only one of them produces a record of why.

Memory corruption sharpens the same distinction. Wrapping a suspect function in a broad exception handler, or simply increasing a buffer's size until the crash stops appearing, restores function with very little diagnostic yield: the program runs, but the actual write past bounds is still happening somewhere, silently, and will resurface in a different form later. A memory sanitizer that halts execution at the exact instruction where the invalid write occurs has comparatively low restorative value on its own, since it does not fix anything, but its discriminatory value is close to maximal, because it collapses the gap between corruption and detection down to a single instruction. The two interventions are not competing solutions to the same problem. One makes the program run. The other makes the fault legible.

Race conditions are the case where the two objectives can come apart most severely, because here the highest-restorative intervention is often also the one that destroys the most information. Adding a global lock around the contested section of code will usually make the bug disappear, in the sense that the program stops exhibiting the symptom, by removing the concurrency that made the bug possible in the first place. This means the underlying invariant violation, whatever thread ordering actually produced the corrupted state, is never identified. The fix and the erasure of evidence are the same act. A narrower intervention, adding an assertion on the specific invariant suspected of being violated and then attempting to reproduce the failure under controlled scheduling, has far lower restorative value, since it may not fix anything at all, but it is the only kind of intervention capable of confirming or ruling out a specific causal hypothesis.

Across all three registers, the pattern is consistent. High-restorative interventions tend to act on the system as a whole, replacing, resetting, or isolating a large region at once, and by acting broadly they average away the very distinctions a diagnosis needs. High-discriminatory interventions tend to act narrowly, on one candidate at a time, and accept a slower or less certain path back to working order in exchange for a legible record of what happened. Neither kind of intervention is more correct than the other in a general sense. What separates a competent repairer from an incompetent one is knowing, in a given moment, which objective the situation actually calls for, and knowing that these are two different objectives rather than one.

5 The Workaround as a Causal Theorem

A workaround is usually treated as an embarrassment, a patch applied in place of a proper fix, tolerated until someone eventually gets around to doing the real work. That framing misses what a workaround actually is. A workaround that reliably restores a property, without anyone ever locating the underlying fault, is not an absence of knowledge. It is a specific and verifiable piece of knowledge, one that simply stops short of full resolution.

Consider a small, extremely common pattern: a service intermittently fails under load, and someone discovers that restarting a particular subprocess every few hours prevents the failure from occurring at all. No one has identified the actual leak, race, or resource exhaustion responsible. The team never gets a root cause. And yet the workaround encodes something precise: under the conditions that produce the failure, this specific intervention reliably restores the property in question. That statement has the same logical shape as a scientific finding. It specifies a state, an intervention, and a consequence, and it is falsifiable in the ordinary sense: if the restart stops preventing the failure, the workaround is wrong, or the conditions have changed, and either outcome is informative.

This is different from a full diagnosis, which would also specify why the intervention works, tracing the restart's effect back through the resource being exhausted to the code responsible for exhausting it. But it would be a mistake to treat the workaround as diagnosis's incomplete or lesser cousin. It occupies a different position entirely. Diagnosis narrows the hypothesis space about mechanism. A workaround narrows the space of effective interventions without ever committing to a mechanism at all, and in systems complex enough that full diagnosis is prohibitively expensive, that is frequently the only kind of knowledge that gets produced.

This gap, between establishing that an intervention works and explaining why, has a formal counterpart in Judea Pearl's account of causal inference, which distinguishes sharply between passively observing a system and actively intervening on it. Pearl's calculus shows that a causal effect can, under the right conditions, be identified rigorously from interventional evidence, what happens to some property when an intervention is deliberately applied, without ever recovering the full causal structure connecting the two. A workaround is an informal, underpowered version of exactly that kind of result. It establishes an interventional fact, under this state, this intervention changes this property, without requiring, or producing, the structural account that would explain the connection. Mechanism, on this view, is not the minimum unit of useful causal knowledge. A single well-confirmed intervention is.

The distinction matters practically because workarounds are not equally trustworthy. A workaround discovered through the same discriminatory discipline described in the previous section, one input changed at a time, one variable isolated, one hypothesis eliminated, carries real evidential weight even without a named mechanism. A workaround discovered by trying several things at once until the symptom disappeared starts out carrying almost no mechanistic weight, because nothing about how it was found rules out coincidence, a second unrelated change, or a condition that will silently stop holding. That is a claim

about causal attribution specifically, not about reliability in general. If the workaround keeps working across many repetitions and varied conditions, it does accumulate genuine evidence that the intervention has some real effect, entirely apart from the question of which effect, through which mechanism, a question the sloppy discovery procedure leaves exactly as open as it was on day one. What a bad discovery process permanently damages is the ability to say why the workaround works. It does not prevent repeated success from being evidence that it does. Two workarounds that produce identical behavior can therefore encode completely different amounts of knowledge, and nothing about the workaround itself, taken as a bare fact, reveals which kind it is. Only the process that produced it does.

This is why a body of accumulated workarounds, the folk knowledge of a team or a codebase, a file of shell aliases, a wiki page of known issues and fixes, a comment reading only "do not remove, breaks build," is not simply a backlog of deferred proper fixes. It is closer to an unindexed collection of causal theorems of uneven provenance, some rigorously earned, some accidental, with no label distinguishing one from the other. Recovering that distinction after the fact, asking of an old workaround not just whether it still works but how it was found, and what that method implies about how much it can be trusted, is itself a diagnostic act, applied one level up.

6 Accumulator Substrates

Every diagnostic act described so far produces information, but information produced is not automatically information retained. Something has to hold the discriminatory result somewhere, so that the next person, or the same person later, does not have to rediscover it from scratch. Call whatever performs that function an accumulator substrate, and notice that different substrates retain different parts of the same event with very different fidelity.

A commit message that says only "fix build" retains almost nothing beyond the bare fact that something was broken and is not anymore. A commit message that says the responsible package was pinned because a specific newer version introduced a breaking change to its exported types retains the discriminating result itself, the specific hypothesis that was confirmed, in a form the next reader can verify or challenge without repeating the search. Both are entries in the same version-control history. Only one of them is doing the job an accumulator is supposed to do.

A comment reading "do not remove, breaks build" is an unusually pure example of the problem, because it retains only the conclusion of a discriminatory process while discarding the process itself. It preserves the fact that some intervention has some consequence, without preserving which of many possible mechanisms explains the connection. The comment is not wrong. It is an accumulator with almost no bandwidth: it stores one bit of restorative information and none of the discriminatory information that produced it. A future maintainer inherits the constraint without inheriting any way to evaluate whether the constraint still applies once the surrounding code has changed.

A well-kept incident postmortem sits at the opposite end from the terse comment. It typ-

ically records not just what fixed the problem but what was tried and ruled out along the way, which is precisely the part a bare fix commit or a terse comment throws away. That ruled-out material has value disproportionate to its size, because it prevents the next person from re-running an eliminated hypothesis, which is one of the most common ways diagnostic effort gets wasted twice.

Human memory is its own accumulator substrate, with a characteristic failure mode different from either of the written forms. An experienced engineer often retains a strong intuition about where a given class of bug tends to live, learned across many discriminatory episodes, without retaining the individual episodes that produced the intuition. This is efficient in one sense, since intuition is cheap to consult and does not require searching a history, and lossy in another, since intuition cannot be audited. When someone attributes a problem correctly to a familiar suspect on instinct, there is no way to check whether that judgment came from many properly discriminated incidents or from one memorable coincidence being over-generalized.

None of these substrates is simply better than the others in the abstract. Each trades retention against cost in a different place: terse commits are cheap to write and expensive to later interpret, postmortems are expensive to write and cheap to later interpret, intuition is nearly free to consult and impossible to verify. What determines whether a system's accumulated repair history is actually useful is not how much diagnostic effort went into producing it originally, but how much of that effort survived in whatever substrate absorbed it. A brilliant piece of discriminatory reasoning, captured in a commit message nobody will ever read carefully, is functionally almost identical to no reasoning having happened at all.

7 Epistemic Compression

What a terse comment does to the episode that produced it has a general name: compression. A full diagnostic episode is a sequence of hypothesis spaces narrowed by successive interventions and observations, $H_0, i_1, o_1, H_1, i_2, o_2, \ldots, H_n$, with each narrowing justified by what was tried and what it ruled out. An accumulator substrate stores some projection of that sequence, and the projection is rarely faithful. "Do not remove, breaks build" compresses the entire episode down to something close to a single admissibility judgment, $i \notin A$, this intervention is not permitted, without the observation history that justified the judgment surviving alongside it. The compressed artifact remains operationally useful long after its justification has disappeared, which is exactly the pattern intuition exhibits at a larger scale: a diagnostic history spanning years collapses into a disposition to check one subsystem before another, and the disposition outlives every individual episode that produced it.

Different accumulator substrates preserve different projections of the same underlying episode, and the dimensions along which they vary are worth naming separately: which outcome resulted, which intervention was applied, the sequence in which candidates were tried, which alternatives were eliminated and how, the causal attribution eventually settled on,

and the confidence that attribution deserves. A terse commit message typically keeps only the first of these. A postmortem, done well, keeps most of them. Intuition keeps something closer to a weighted average across many outcomes, with almost none of the rest.

Put in its strongest form: an organization does not, in any very meaningful sense, possess knowledge about its own systems. It possesses retained discriminations, unevenly distributed across whichever substrates happened to catch them. What gets called institutional knowledge is usually the residue of many past hypothesis eliminations, preserved with different fidelity in commit history, comments, incident reports, internal conventions, and the intuitions of whoever has been debugging the system longest. When those substrates lose their provenance, an organization can go on behaving correctly while becoming genuinely ignorant of why that behavior is correct, and the two states, correct and understood, can drift apart for a long time before anything forces them back together.

Mark Wilson has argued, mainly with respect to physics and engineering, that a great deal of working technical knowledge survives not as one unified theory but as a patchwork of locally valid rules, each reliable within some bounded region and stitched to its neighbors by convention rather than derivation, with practitioners routinely getting correct results while avoiding the deeper theoretical unification that would make every patch provably consistent with every other. A mature codebase's accumulated compressions look like a small instance of the same structure. Its dependency pins, odd conditionals, defensive checks, retry loops, feature flags, and initialization orderings function as regionally valid patches, each a compressed record of some earlier failure, correct within whatever narrow envelope produced it and never assembled into a single account of why the system as a whole behaves the way it does. What Wilson calls physics avoidance in engineering has a direct counterpart here: a system's maintainers can go on avoiding the underlying explanation indefinitely, so long as the patchwork of locally compressed constraints keeps holding.

The clearest illustration is the kind of ordering constraint every long-lived codebase eventually accumulates: initialize this subsystem before that one, or a race introduced years earlier reappears. The engineers who diagnosed it originally are usually gone, and the reasoning behind the ordering is rarely written down in any form still consulted. What remains is the constraint itself, enforced by a comment, a test, or simply the order the code happens to be written in. The system continues to behave correctly with respect to a failure mode that no one currently on the team can explain. This is not a claim that the system knows anything in the sense a person knows something. It is a narrower claim about where the information ended up: encoded in which transformations the system's current form permits and forbids, rather than in any proposition available to be stated by a person. A codebase can retain causal knowledge after every person who once possessed the explanation for it has left.

Put in the most direct terms available: a mature software system is partly composed of compressed failed experiments. Its dependency pins, odd conditionals, defensive checks, retry loops, feature flags, and ordering constraints exist, in large part, because some earlier continuation of the system failed, and what survives is the residue of that failure after its diagnostic history was compressed down to an operational rule. This gives a sharper

account of what is usually called technical debt than the standard description of a quick fix that will cost more later. Some of what gets called debt is not structurally expensive at all; a workaround can be efficient and entirely harmless to a system's performance. What makes it expensive is that its causal justification has been compressed below the threshold needed for reconstruction. The future maintainer inherits the intervention without the evidential history that would let them safely remove or extend it. Some technical debt, in other words, is epistemic debt rather than structural debt, and paying it down means rediscovering a justification, not rewriting code.

8 Close

A working system withholds almost everything about its own structure, because too many internal arrangements are compatible with the single fact of it currently functioning. Failure is not simply the loss of that function. It is the release of information that working conceals, information about dependency, about sequence, about which parts of the system actually mattered and which were incidental.

But that information is not handed over freely. It has to be extracted through intervention chosen for what it will distinguish rather than what it will fix, and once extracted, it has to be caught by some substrate capable of holding it, or it dissipates as quickly as it appeared. A workaround that works is a small piece of that information, committed to action without ever being committed to explanation. An accumulator that only remembers outcomes and not the process that produced them is a leak in a system's capacity to learn from its own failures, no less real for being invisible.

A codebase with twenty years of accumulated repairs behind it is not automatically twenty years wiser about itself. It is only as knowledgeable as whatever its accumulator substrates managed to preserve. Understood this way, the deepest product of a repair is not the corrected state it leaves behind but the changed space of future possibilities that corrected state makes admissible. A repair has fully entered a system's history only when some distinction learned from its failure survives strongly enough to constrain what the system, or whoever maintains it, does next.

The question a repairer learns to ask, across dependency graphs, corrupted memory, and racing threads alike, is not really what is broken. It is closer to: what is the smallest change whose result will reveal the most about what currently cannot be seen, and where will that result go once it has been found.

Put most simply, the opposite of ignorance in a system like this is not explanation. It is successful discrimination, the accumulated ability to tell apart states that used to look identical. Explanation, when it eventually arrives, is a further achievement built on top of that ability, not a substitute for it, and a great deal of expert practice runs entirely on the ability alone.

A sufficiently old codebase is not merely a program. It is a fossilized record of everything that was once allowed to go wrong, and, wherever its accumulator substrates did their job,

of what its maintainers learned well enough to keep from going wrong the same way twice.

References

[1] Feyerabend, P. (1975). *Against Method*. London: New Left Books.

[2] Pearl, J. (2009). *Causality: Models, Reasoning, and Inference*, 2nd ed. Cambridge: Cambridge University Press.

[3] Wilson, M. (2006). *Wandering Significance: An Essay on Conceptual Behaviour*. Oxford: Oxford University Press.