

Executive Guide to Self-Improving Agent Harnesses

A reference for engineering leaders who already run agents in production. It defines the four components of a self-improving harness, the instrumentation and evaluation discipline that keeps improvement honest, and the promotion gate that stops a system from getting worse while it gets faster.

01

The Harness Is the Product

02

What Self-Improvement Actually Means

03

The Four Components of a Self-Improving Harness

04

Instrumentation: Trajectories Worth Learning From

05

Evaluation: A Score That Does Not Drift

06

Memory That Outlives the Session

07

The Promotion Gate: Change Is a Release

08

Failure Modes and How to Detect Them

09

Evaluation Checklist

Most enterprises running agents in production are still tuning the wrong variable. When an agent fails at a task a competent person would have finished, the reflex is to reach for a newer, larger or more expensive model. That reflex is almost always the least productive response available. The model is one input into a far larger program that decides what the agent sees, what it remembers, which tools it may call, how its work is scored, and whether any proposed change to that program reaches production. That program is the harness, and it is where most of the variance in agent outcomes lives.

This guide is for the CTO, the VP of engineering and the AI platform lead who already run agents in production and are being asked why results are uneven. Our position is plain. Harness quality, not model choice, is the dominant variable in agent outcomes, and a self-improving harness is a buildable engineering artifact rather than a research aspiration. It has named components, interfaces and a release process, and a platform team can operate it without standing up a research lab.

The governance expectations around that artifact are becoming explicit. The [NIST AI Risk Management Framework](#) is a voluntary framework published in January 2023, and therefore more than eighteen months old as of 2026, intended to help organizations incorporate trustworthiness considerations into the design, development, use and evaluation of AI systems. A harness that cannot be measured, scored against a stable baseline or rolled back is hard to defend under any framework of that shape. The rest of this guide describes the harness that can be defended.

SECTION 01

The Harness Is the Product

It is worth being precise about the term. The 2026 research paper [MESH-Harness](#) defines an agent harness as the code that organizes context, maintains state and coordinates tool calls for a language model, and studies how to improve that harness while model weights stay fixed. That last clause is the important one. Improve the harness and hold the model constant, and you are measuring your own engineering. Swap the model and hold the harness constant, and you are measuring someone else's.

Where the variance comes from

Consider what differs between two runs of the same agent on the same task. The prompt template, the retrieved context, the tool schemas, the retry policy, the available memory, the temperature, the environment state left by the previous run and the grader that judged the result all vary. Of those, the model checkpoint changes least often in a mature system. Everything else changes weekly, and every one of those changes is made by your team, in your repositories, under your review. Deciding that the model is the lever, when the lever you actually pull is the harness, is a category error about where your work happens.

01

The Harness Is the Product

02

What Self-Improvement Actually Means

03

The Four Components of a Self-Improving Harness

04

Instrumentation: Trajectories Worth Learning From

05

Evaluation: A Score That Does Not Drift

06

Memory That Outlives the Session

07

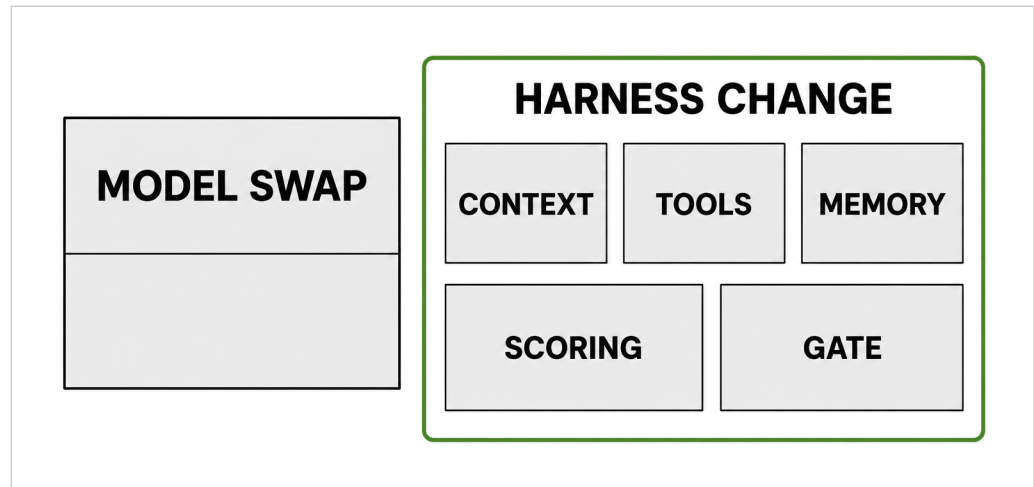
The Promotion Gate: Change Is a Release

08

Failure Modes and How to Detect Them

09

Evaluation Checklist



Why this is good news

The claim that **HARNESS QUALITY IS THE DOMINANT VARIABLE** is not a criticism of model vendors. It is an argument about control, and it is the more hopeful half of the story. The harness is the part of the system you own end to end: its source is in your repositories, its behaviour is in your traces, its failures are in your incident history and its improvement is in your release pipeline. It can be specified, built, tested and governed with the discipline you already apply to any distributed system. That is what makes the self-improving harness a build target rather than a research programme.

Treat the harness as a first-class product with an owner, a version, a test suite and a release note. Without those, the outcome variance you are seeing is not a mystery. It is an unmanaged dependency.

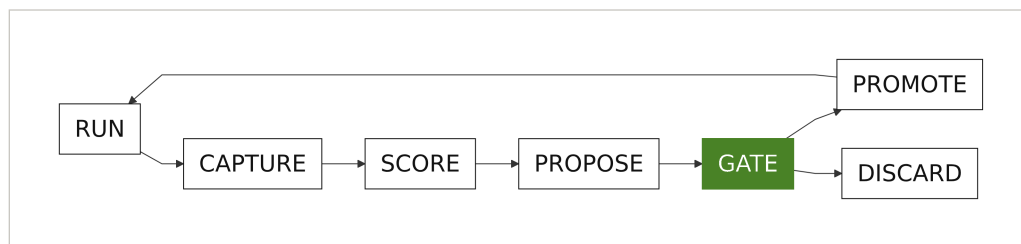
SECTION 02

What Self-Improvement Actually Means

Self-improvement is a specific mechanism, not a mood. We use the phrase to mean that the harness changes as a consequence of evidence generated by its own production runs, and that those changes pass through a gate before they affect users. Three properties follow. The evidence must come from real work rather than from imagination. The change must be proposed against a versioned baseline rather than applied in place. The gate must be able to say no, and the no must be durable, because a rejected change that returns next cycle is not a gate but a delay.

What it is not

Self-improvement is not fine-tuning, and it is not an engineer editing prompts by hand after reading a complaint. We would rather have that maintenance than not, but it does not scale and it does not compound. The difference is that a self-improving harness runs the loop on itself with a measurable acceptance criterion. A harness that stores a summary of yesterday's conversation is not self-improving. A harness that proposes a change to its own skill document, measures that change on items it did not invent last night, and then promotes or discards it, is. The distinguishing feature is a scored, gated proposal.



The run produces the evidence. Capture preserves it in a form specific enough to reason about later. Scoring attaches a number that means this week what it meant last week. Proposal turns scored evidence into a concrete candidate change, whether a prompt edit, a tool schema revision, a retrieval policy or a memory consolidation rule. The gate decides, promotion ships, and a rejection goes to discard rather than to a backlog, because an unreviewed backlog of rejected proposals becomes an unaudited shadow of the live configuration. Teams that reach production with this loop usually fail at the gate, and they fail the same way: the aggregate score rises, the change ships, and items the harness already handled correctly start failing quietly.

The gate is the innovation. Any competent team can generate candidate improvements; the hard engineering is refusing to ship the ones that only look better.

Flexivan arrived at this conclusion from the operations side, and described what changed once the loop rather than the model was held accountable:

“AI used to be experimental at FlexiVan. It is no longer experimental. It is operational.”

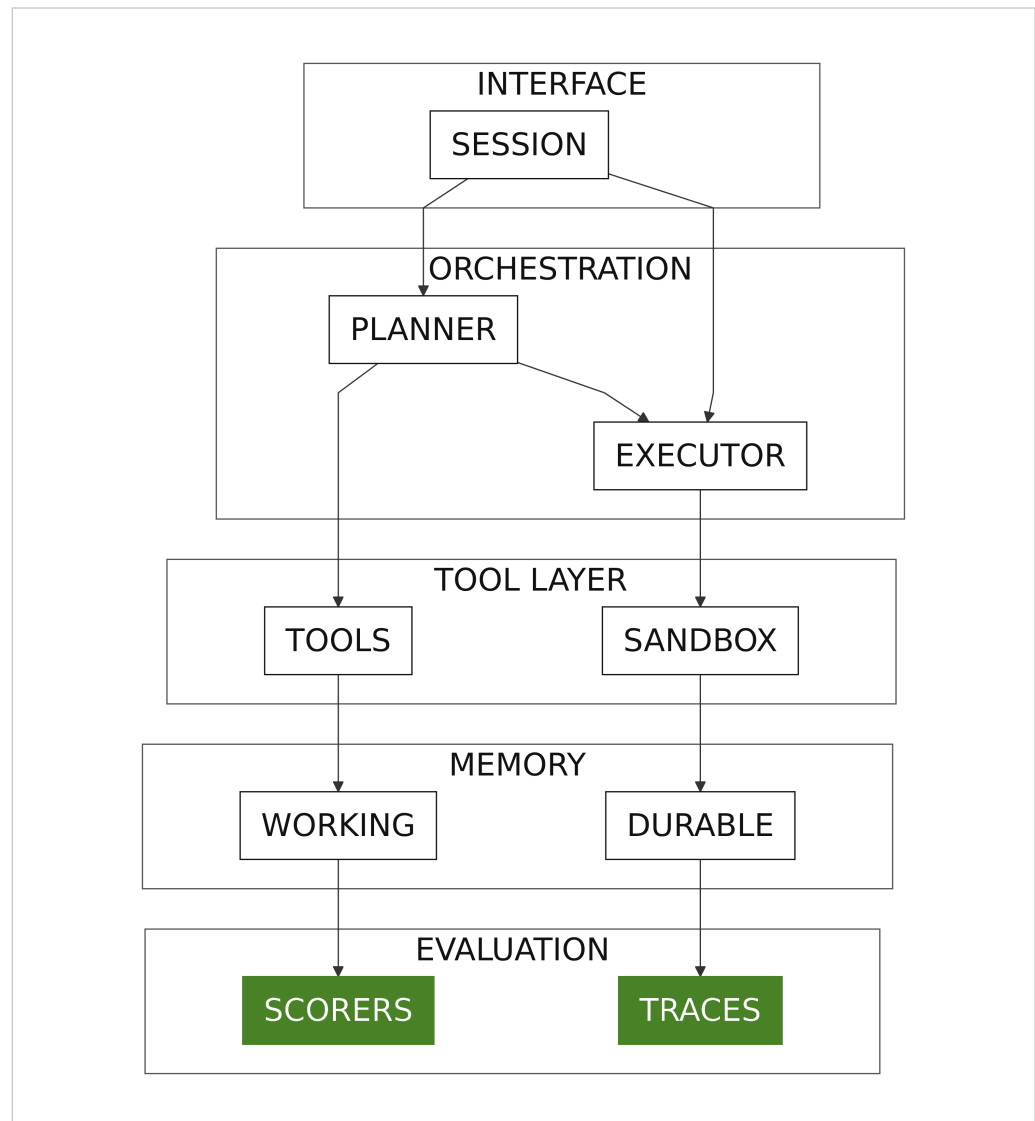
Sagar Chikkala
Chief Information Officer @ Flexivan

FLEXIVAN™

SECTION 03

The Four Components of a Self-Improving Harness

A self-improving harness is not one program. It is four components with narrow interfaces, and each exists to make the other three trustworthy. Trajectory capture produces the raw evidence. A non-drifting evaluation set produces the score. Durable memory stops the system relearning what it knew. The gated promotion path decides what reaches users. Remove any one and the other three degrade in a nameable way: capture without evaluation produces data nobody trusts, evaluation without memory forgets its own corrections, memory without a gate produces drift with provenance, and a gate without capture produces decisions made on opinion.



It helps to see the harness as a stack rather than a script. An interface layer accepts work. An orchestration layer decides the plan, the step order and the stopping condition. A tool layer defines what the agent can do and with whose permissions. A memory layer decides what persists and what is retrieved. An evaluation layer observes all of it and produces the score. In a mature harness each layer is versioned separately, and the promotion gate can move any one of them independently.

What the model owns and what the harness owns

The division of labour between model and harness is the most useful boundary to draw explicitly in an architecture review, because teams routinely try to fix harness problems with model budget.

DIMENSION	WHAT THE MODEL OWNS	WHAT THE HARNESS OWNS
Determinism	Sampling behaviour inside a single call	Temperature and seed policy, retries, and the reproducibility of everything around the call
Cost	Per-token price and context window	Context budgeting, caching, model routing, tool fan-out, and the spend ceiling per task
Tool access	The ability to emit a call in a valid schema	Which tools exist, their permissions, their schemas, their blast radius and their rate limits
Memory	Nothing beyond the current context window	What is written down, where it lives, how it is retrieved, consolidated and invalidated
Evaluation	Raw capability on the task	Task selection, graders, thresholds, the baseline score, and the promote-or-reject decision
Failure recovery	Graceful degradation within one response	Retries, checkpointing, escalation, compensating actions and abort criteria

Table: What the model owns versus what the harness owns. The left column is largely a vendor relationship; the right column is your engineering surface.

Reading down the right-hand column is a job description, and every entry is an engineering decision with an owner. Reading down the left-hand column is a vendor relationship. If a failure maps to the left column you may need a different model. If it maps to the right you need a harness change, and no amount of inference spend will produce it.

Loblaw Digital reached a similar split from a platform-engineering standpoint, and described what the separation of duties meant for their teams:

“Shakudo's platform allows our teams to focus on solving real problems rather than reinventing core plumbing. It enables us to scale our agentic AI capabilities responsibly across the organization.”

Charu Pujari

Senior Vice President, Engineering and AI, Loblaw

LoblawDigital

Draw the model line and the harness line in writing, and put a name against every row in the harness column. Ownership is what turns a stack diagram into a roadmap.

SECTION 04

Instrumentation: Trajectories Worth Learning From

Everything downstream depends on whether the trajectory you stored is specific enough to reason about a month later. Most teams begin by logging the conversation and the final answer, then discover that this record cannot explain anything. It does not show which tool returned what, which retrieved passage the model actually used, how long each step took, what the environment looked like before the run, or what the agent's subprocesses did. A trajectory that records only the agent's own account of its actions is a summary written by the party under audit.

The corrective is to treat a trajectory the way production systems have treated distributed traces for a decade: a trace is a set of spans, each with a name, a duration, a status and attributes,

01	The Harness Is the Product
02	What Self-Improvement Actually Means
03	The Four Components of a Self-Improving Harness
04	Instrumentation: Trajectories Worth Learning From
05	Evaluation: A Score That Does Not Drift
06	Memory That Outlives the Session
07	The Promotion Gate: Change Is a Release
08	Failure Modes and How to Detect Them
09	Evaluation Checklist

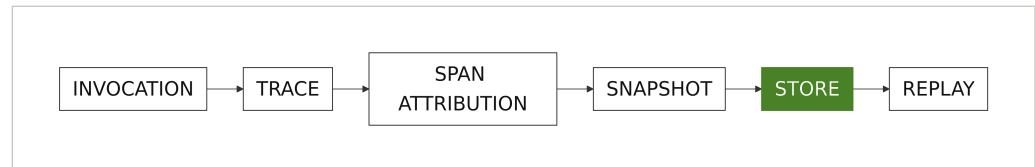
joined into one causal structure. In 2026 this is no longer a bespoke design problem. The [Gen AI semantic conventions published by OpenTelemetry](#) define a standard attribute set for agent and model spans, covering the agent identifier and version, the operation name, the conversation identifier, input and output messages, token usage and evaluation scores. Adopting the standard puts agent traces into the same backend, queried the same way, as the rest of the estate.

The signals every trajectory must capture

Within that span structure we require the following signals from every production trajectory. A record missing any of them will fail you at exactly the moment you need it.

- The invocation record: who or what asked for the work, the task identifier and the harness version, so every score traces back to a release.
- The full message array in order: system prompt, user turns, model responses and reasoning content, because the ordering is often the bug.
- Every tool call as its own span: name, arguments, result, duration, error and the permission it ran under.
- Every retrieval and memory read: the query, the identifiers returned, the passages that entered context and the scores that ranked them.
- Environment state before and after: files touched, records changed, external calls made, and the snapshot that makes replay possible.
- The outcome measured in the environment, plus the grader verdict: which checks ran, which passed, and the human review where one happened.

From capture to replay



If a run cannot be replayed, it cannot be an eval item. Design the snapshot for replay on day one, because retrofitting determinism into a live environment is the hardest part of this build.

SECTION 05

Evaluation: A Score That Does Not Drift

An evaluation set is useful only if the same harness revision scores the same way next month as it did this month. That property is not free, and the most common way to lose it is to edit the eval set in the same change that edits the harness. Anthropic's January 2026 guide to [demystifying evals for AI agents](#) draws the vocabulary precisely: a task defines the work, a trial is one execution of it, a grader scores some aspect of performance, and the outcome is the final state in the environment. A booking agent can report that a flight was booked, while the outcome is whether the reservation exists in the database. That distinction is the basis of a score you can trust.

The same guide supplies the second half. Trials must start from a clean environment, because shared state between runs produces what look like capability failures but are infrastructure flakiness, and shared state can also inflate performance: their teams observed an agent gaining an advantage by reading the git history left behind by earlier trials. An evaluation environment that leaks state is measuring the leak.

01

The Harness Is the Product

02

What Self-Improvement Actually Means

03

The Four Components of a Self-Improving Harness

04

Instrumentation: Trajectories Worth Learning From

05

Evaluation: A Score That Does Not Drift

06

Memory That Outlives the Session

07

The Promotion Gate: Change Is a Release

08

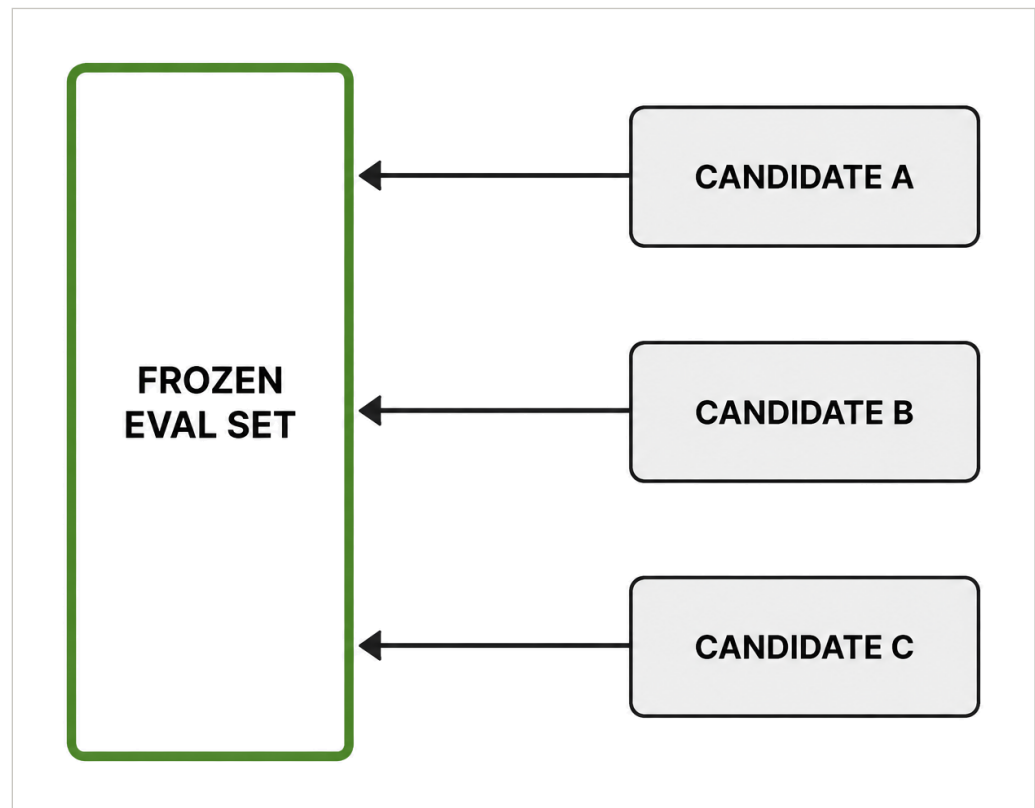
Failure Modes and How to Detect Them

09

Evaluation Checklist

Properties of a trustworthy evaluation set

- Representativeness: items come from the distribution of work the harness actually receives, including the boring majority and the rare expensive failure.
- Outcome grounding: every item is judged on the state of the environment after the run, not on the agent's claim of success.
- Isolation: every trial starts from a clean, pinned environment, so no result is contaminated by the run before it.
- Pairability: the same item can run against baseline and candidate with identical inputs, so the comparison is per item rather than on average.
- Versioning: the eval set carries its own version and does not change inside a promotion window.
- Adversarial coverage: the set includes items designed to trigger known failure modes and prohibited actions, not only the happy path.
- Ownership: a named person decides what enters the set and reads the trajectories when a score moves unexpectedly.



Scoring discipline

Two decisions matter more than the choice of grader. Score the outcome in the environment, and use the trajectory only for attribution. Then compare an item to itself: a per-item paired comparison, in which the current harness and the candidate run on identical items, exposes regressions that an aggregate mean hides, and protects against the upward bias you get when you select the best observed score from a finite and noisy validation set. We distrust any rule that keeps a change simply because the average improved, because a mean can rise while individual solved items break.

01

The Harness Is the Product

02

What Self-Improvement Actually Means

03

The Four Components of a Self-Improving Harness

04

Instrumentation: Trajectories Worth Learning From

05

Evaluation: A Score That Does Not Drift

06

Memory That Outlives the Session

07

The Promotion Gate: Change Is a Release

08

Failure Modes and How to Detect Them

09

Evaluation Checklist

```
def score(transcript: Trajectory, outcome: EnvironmentState, rubric: Rubric) ->
Grade:
    # one scorer per eval item, deterministic given the environment snapshot
    checks = rubric.checks_for(outcome)
    return Grade(passed=[c for c in checks if c.evaluate(transcript, outcome)],
                  failed=[c for c in checks if not c.evaluate(transcript, outcome)])

promote = (
    zero_regressions                                # no item the baseline
    solved may break
    and score_delta > noise_band(n=len(items), z=1.96) # improvement must beat run-
    to-run noise
    and zero_guardrail_violations                    # prohibitions are absolute
)
```

Saturation

An eval set decays even when nobody edits it. A suite that a candidate passes completely still tracks regressions but provides no signal for improvement, and the Anthropic guide is explicit that saturation at 100 percent leaves no room to learn. Set a retirement rule at design time: when an item has passed for several consecutive releases, move it into a frozen regression set whose only job is to catch breakage, and pull a harder held-out task into the active set.

A SCORE THAT MOVES WITH THE HARNESS IS NOT A SCORE. Version the eval set separately, freeze it during a promotion window, and treat the regrading of an existing item as a harness change that must itself pass the gate.

SECTION 06

Memory That Outlives the Session

A harness without durable memory relearns the same lessons forever. Every session starts from nothing, every correction a user makes is lost at the end of the conversation, and the same failure recurs in the same way next week. Memory is how a harness converts a solved problem into a permanent one.

Memory is also where teams most often build something worse than nothing. The 2026 research on [GenMem, a generative symbolic memory design for self-evolving harnesses](#) frames the problem: long-term memory supports self-evolution by retaining experience and skills across tasks so they can be retrieved, reused and revised, yet only a small and task-dependent subset of trajectories warrants retention. Writing everything down is not memory. It is a retrieval problem handed to your future self, and the more you write, the more the address of the useful item matters.

Three tiers

We separate memory into three tiers with different lifetimes and different failure modes. Working memory is the context window of the current task. It is the only tier the model reads, it is bounded, and it should be assembled for the task at hand rather than accumulated. Episodic memory is the record of what happened: trajectories, outcomes and corrections, tagged with the harness version and the environment that produced them. Semantic memory is the distilled reusable form, the skill document, the tool-use rules, the decision logic and the exceptions, which is the artifact a self-improving harness actually edits.

01

The Harness Is the Product

02

What Self-Improvement Actually Means

03

The Four Components of a Self-Improving Harness

04

Instrumentation: Trajectories Worth Learning From

05

Evaluation: A Score That Does Not Drift

06

Memory That Outlives the Session

07

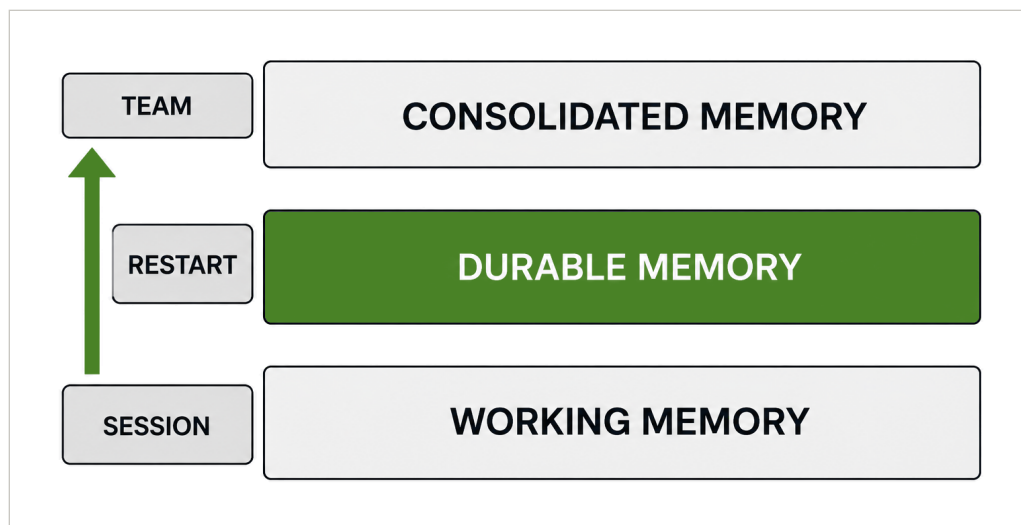
The Promotion Gate: Change Is a Release

08

Failure Modes and How to Detect Them

09

Evaluation Checklist



What makes memory durable

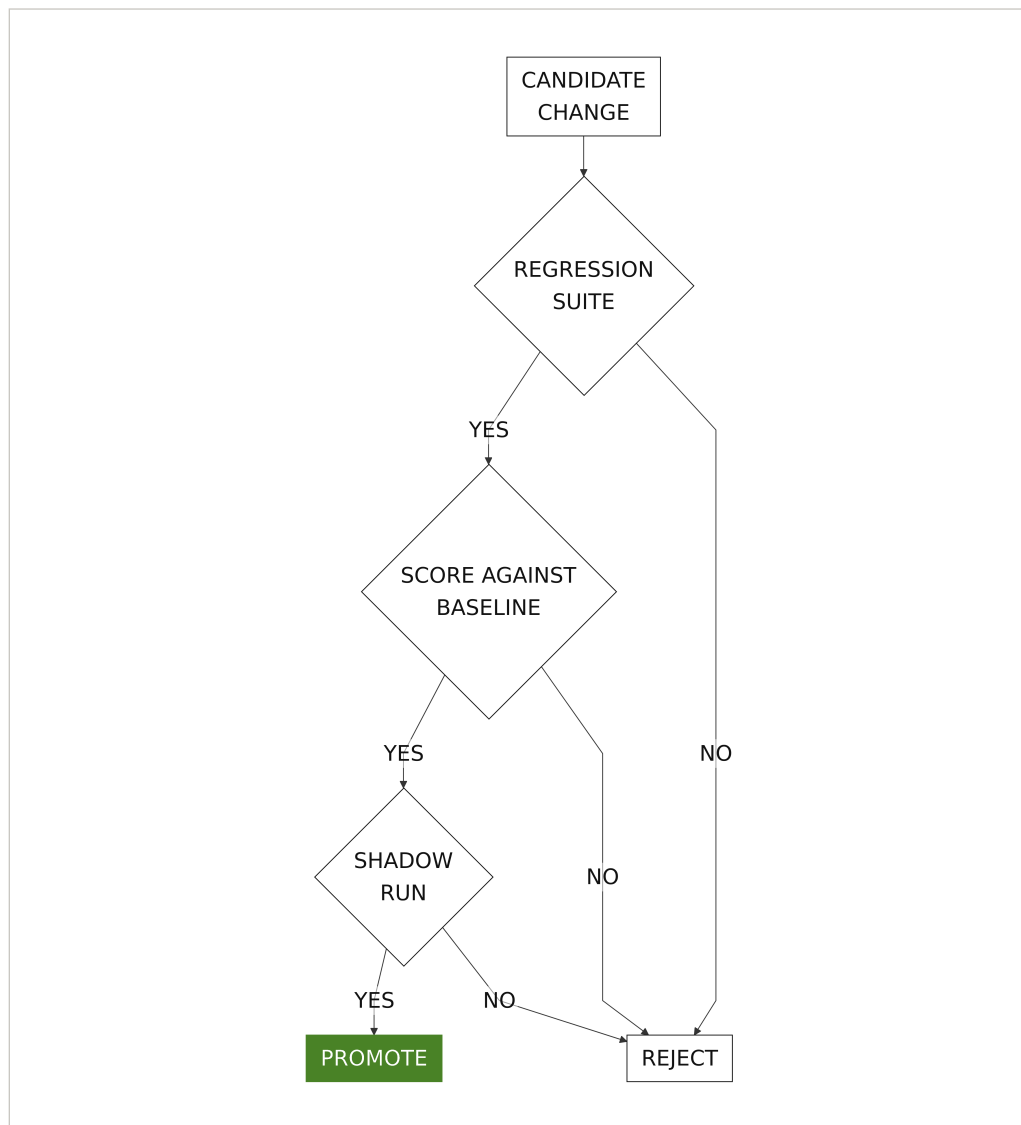
Three properties separate memory that helps from memory that poisons. Provenance: every write records the run that produced it, the harness version and the outcome, so a retrieval decision can be audited and a bad memory traced to its source. Expiry: memory expires by default, and the burden of proof falls on keeping it, because a lesson learned in an environment that has since changed is worse than no lesson at all. Invalidation on change: when a tool schema, a policy or an environment contract changes, the memory that depended on it is invalidated in the same release rather than discovered by a failing retrieval six weeks later.

Memory is a capability with a blast radius. Treat every write as a production change: provenance required, expiry by default, and invalidation coupled to the release that breaks it.

The gate is where self-improvement becomes an engineering discipline or becomes a liability. Our principle is simple: a change to the harness is a release, and it earns the ceremony of a change to a payment service. That places the design inside release engineering rather than inside machine learning. The Google SRE Workbook, whose chapter on canarying releases was published in the 2018 edition and is therefore well over eighteen months old as of 2026, defines canarying as a partial and time-limited deployment of a change in a service and its evaluation. That definition transfers to harness changes unmodified, including the requirement that a canary be time limited and evaluated rather than merely deployed.

The Promotion Gate: Change Is a Release

The decision path



Every stage has a binary exit and the exits are not equivalent. A no at the regression suite is free: the candidate never touched a user, and the failing items return to the proposer as evidence. A no at the score comparison means the change is not an improvement by the standard fixed in advance. A no at the shadow run is the most valuable outcome available, because the candidate was good enough to reach real traffic in a non-authoritative role and still lost. Shadow execution is the harness equivalent of a flight simulator, and the only way to see how a change behaves on traffic nobody curated.

Gate signals

SIGNAL	THRESHOLD	ACTION	ROLLBACK
Regression suite pass rate	No item the baseline solves may fail	Block promotion	Not required, the candidate never shipped
Aggregate eval score	Must exceed the baseline by more than the measured noise band	Promote to shadow	Revert the harness revision
Per-item paired comparison	Zero regressions across identical items	Return the failing items to the proposer	Not required
Shadow disagreement rate	Within the share agreed when the experiment was designed	Hold and re-run with a larger sample	Revert the harness revision
Latency and cost per task	Inside the task budget agreed at design time	Hold for a cost review	Revert the harness revision
Guardrail violations	Zero, with no tolerance	Abort the candidate and open an incident	Revert and freeze the promotion path

Table: The promotion gate, read row by row. Statistical and safety signals sit inside one decision so that no single aggregate number can carry a change into production on its own.

Controls required before autonomy increases

The gate can be operated by a human at first, and for most teams in 2026 it should be. The question every platform team eventually asks is when the gate itself may be automated. Our answer is that autonomy is extended one control at a time, in this order, and each increment is earned by the previous one running cleanly.

- Automated rollback. Reverting any promoted component is a single versioned action with a rehearsed path, proven in a drill, before anyone proposes that the gate run unattended.
- Shadow execution at scale. Candidates run against a representative share of real traffic without reaching a user, so decisions rest on production behaviour rather than curated items.
- Hard guardrails outside the harness. Prohibited actions are enforced by permissions and policy in the tool layer, not by instructions in a prompt, so a wrongly promoted change cannot reach a prohibited system at all.
- Cost and rate ceilings. Per-task and per-window spend and call limits are enforced by the platform, so an accepted change cannot convert a quality gain into an unbounded bill.
- A complete audit trail and a named owner. Every promotion records the candidate, the evidence, the thresholds and the resulting harness version, and one team is accountable for the harness over time.
- A freeze path. There is a documented way to stop the loop, hold the current version indefinitely and operate the harness by hand while a problem is investigated.

A PROMOTION IS A RELEASE. If a harness change cannot be described in a release note, attributed to an owner and reverted in one action, you are not running a promotion gate. You are running a hope.

Failure Modes and How to Detect Them

Self-improving harnesses fail in ways that look like success. That is the defining hazard of the pattern, and it is why the detection controls here matter more than the improvement mechanism.

The most instructive 2026 result on this point is research into [compositional safety failures in harness evolution](#). Harness evolution means continually updating persistent components such as memory, prompts, skills and tools. The paper finds that interactions among individually safe and utility-preserving component updates can produce unsafe agent behaviour, and it reports 43 pairwise and 18 irreducible three-way compositional safety failures identified across three safety benchmarks. The implication is uncomfortable: validating each candidate update on its own is not sufficient, because the risk lives in the combination.

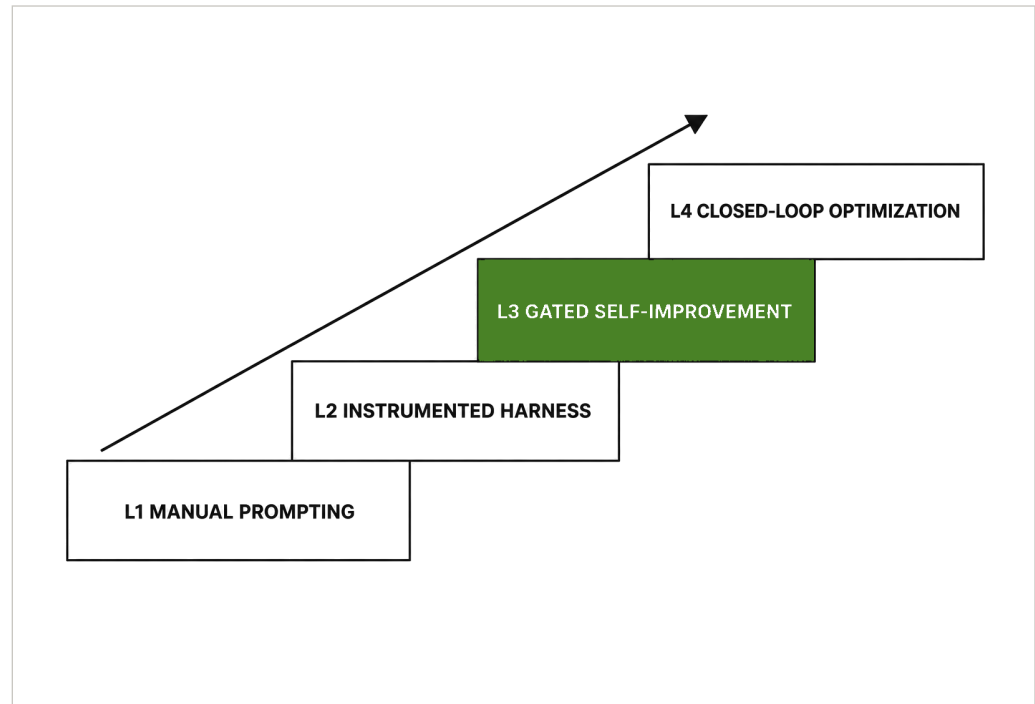
That reshapes how you scope a regression suite. If two changes are each safe and the pair is not, a gate that evaluates one change at a time passes both. Either validate combinations, or serialise promotions so that at most one persistent component changes per window, or do both at different maturity levels.

A detection table

FAILURE MODE	SYMPTOM	CONTROL
Eval set drift	Scores climb while production complaints do not fall	Version the eval set with the harness and freeze it during a promotion window
Eval saturation	The suite passes at its ceiling and no longer separates candidates	Retire solved items and rotate in harder held-out tasks
Metric gaming	A score improves through a shortcut rather than the intended capability	Grade the outcome in the environment and read the trajectories
Silent regression	The aggregate score rises while previously solved items break	Per-item paired comparison against the baseline
Memory poisoning	One bad episode is retrieved long after the context that produced it expired	Provenance on every write, expiry by default, invalidation on change
Compositional safety failure	Individually safe component updates combine into unsafe behaviour	Validate cross-component interactions, not only individual updates
Undetectable attribution	Nobody can explain why a score moved	Span-level attribution from the invocation down to the tool call

Table: Failure modes, their observable symptoms, and the control that catches each one. Every row is something to instrument before you automate anything.

01	The Harness Is the Product
02	What Self-Improvement Actually Means
03	The Four Components of a Self-Improving Harness
04	Instrumentation: Trajectories Worth Learning From
05	Evaluation: A Score That Does Not Drift
06	Memory That Outlives the Session
07	The Promotion Gate: Change Is a Release
08	Failure Modes and How to Detect Them
09	Evaluation Checklist



Tier one runs on individual skill: prompts live in code or in config, results are judged by whoever asked, and there is no shared evidence. Tier two has instrumented the harness: trajectories are captured, a small eval set exists and scores are visible across the team, but changes are still made by hand and nothing is gated. Tier three is the subject of this guide: evidence flows into proposals, proposals face a fixed gate, and promotion is a release. Tier four closes the loop further, letting the system select and sequence its own optimization work under the same gate, and it is the first tier at which the machinery rather than the team sets the pace.

Measure your tier honestly before you automate anything. A gate built on evidence nobody trusts is worse than no gate, because it launders opinion into a number.

SECTION 09

Evaluation Checklist

This section is for the reviewer rather than the builder. The sequence below is the ninety-day plan we would run with a team that already has agents in production. The checklist after it is what we would ask for at the end of those ninety days.

A ninety-day build sequence

1. Weeks one and two: name a single owner for the harness and write down the model-versus-harness boundary, with an accountable engineer against every harness row.
2. Weeks one and two: pick one production agent with real volume and a measurable outcome. Start where a number already exists, not with the hardest or the newest agent.
3. Weeks three and four: instrument that agent, emitting spans under the OpenTelemetry conventions, and confirm that a stored run can be replayed against a pinned environment.
4. Weeks three and four: define the outcome in the environment rather than in the conversation, and prove that a query can produce it. If no such query exists, build it first.
5. Weeks five and six: promote thirty to fifty captured trajectories into a versioned eval set and freeze it, then record the baseline score and its noise band by running the baseline twice.

6. Weeks five and six: write the first scorer and wire it into the pipeline you already have, resisting the urge to build a new evaluation service in this window.
7. Weeks seven and eight: implement the gate with humans in the loop, with every threshold fixed before any candidate is generated.
8. Weeks seven and eight: run one deliberately easy candidate through the full path to promotion and to a one-action rollback, so the mechanics are proven on something that does not matter.
9. Weeks nine and ten: add durable memory with provenance and expiry, limited to one tier and one purpose, and keep memory out of production decisions in this window.
10. Weeks nine and ten: operate the loop by hand for two full cycles and find out where the evidence is insufficient.
11. Weeks eleven and twelve: automate proposal generation only, keep the gate human, and measure the rejection rate. A rejection rate near zero means the gate is not doing work.
12. Weeks eleven and twelve: report tier, coverage and rejection rate to the executive sponsor in the vocabulary of section 8, and publish the first guardrail set.

One request. Fund the harness, not the model, and give the harness an owner. Every remaining item on this list is engineering you already know how to do.

What a reviewer should verify

- ☐ A named owner exists for the harness, distinct from the owner of the model relationship, and that owner signs the release notes.
- ☐ The model-versus-harness boundary is documented, and every harness row has a named accountable engineer.
- ☐ Production trajectories capture per-step tool, retrieval, memory, cost, latency and environment detail, not only messages and final answers.
- ☐ Stored trajectories can be replayed against a pinned environment, and a replay has been demonstrated on a candidate harness version.
- ☐ The eval set is versioned separately from the harness, was frozen during the most recent promotion window, and has a documented entry rule and a retirement rule.
- ☐ Grades are computed from environment outcomes, and trajectories are used for attribution rather than as the basis of the score.
- ☐ Rollback of any promoted harness component is a single versioned action, and it has been rehearsed rather than merely documented.
- ☐ Durable memory writes carry provenance and an expiry, and memory is invalidated in the same release that changes the contract it depended on.
- ☐ An immutable audit trail links every promotion to its candidate, its evidence, its thresholds and the resulting harness version.

Run your data and AI stack on infrastructure you control

Shakudo is the operating system for data and AI. It installs into your own cloud account or on-premise hardware, integrates the tools your team already wants to use, and keeps them upgraded, secured and observable. Platform teams ship products instead of maintaining plumbing.

Book demo

Read customer case studies

WHAT SHAKUDO GIVES YOU

Your infrastructure, your data. Shakudo runs inside your cloud account, your VPC or your data centre. Data and models never leave the boundary you already trust.

Over 200 integrated data and AI tools. Best-of-breed components, installed, version-matched and upgraded for you, with no vendor lock-in on any single layer.

Governance built in, not bolted on. Access control, audit trails and per-workload cost visibility come with the platform rather than as a later project.