Plan, Act, Verify, Repeat

Part III of *The End of Software Engineering and the Rise of Agentic Engineering*

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com

September 2026

Abstract

Agentic engineering runs on a loop: state an intended change, act on it, verify the result, and decide whether to stop, repeat, or escalate. We ask how work gets done and checked in that loop, what counts as verification evidence, and which failure modes recur. We define the loop, the gate, the review-fix loop, bounded iteration, and evidence before assertion, present the loop as a bounded composition of three steps behind a gate, and read two code bases and one hosted product against those definitions. The loop's components were published between 2022 and 2023, and the configuration that literature converged on, the producer critiquing its own work, is the one a gate must exclude: GPT-4 identified its own outputs with 73.5 percent accuracy, and self-preference rises linearly with self-recognition. Every system we examined declares an iteration bound, from two repair attempts to ten outer iterations and a 59-minute session cap, and none reports how many rounds real tasks consume. One system records an exhaustion outcome that the consuming program never reads and defaults a missing verdict to a passing one. We catalogue eleven recurring failure modes, each with a detection test and a remedy, and propose four loop metrics, three unreported by any located source and one measured only indirectly. Whether verification is cheaper than authoring at equivalent quality remains unmeasured, because no source measures both sides on the same tasks. The repository evidence is first-party and establishes that a mechanism exists, not how often it occurs.

1 Introduction

A practitioner who delegates work to an agent has given up the thing that used to provide continuous feedback. When a person writes code, the writing and the checking are interleaved so finely that they are hard to separate: a compile error arrives seconds after the mistake, a failing test arrives minutes after it, and the author's own reading of the code they just wrote arrives in between. Delegating the writing removes that interleaving. What remains is a cycle with a coarser grain, in which a batch of work is produced and then examined, and in which the examination has to be arranged on purpose because nothing arranges it by default. The arrangement raises three questions, RQ3 of the series: how work gets done and checked in a plan, act, verify, repeat loop, what counts as verification evidence, and which failure modes recur.

Part I establishes the series' central claim: software engineering is being redefined as agentic engineering. Its unit of work is moving from hand-authored code to specifications, contracts, orchestrated agent activity, and verification of machine-produced artifacts. The loop studied here is the control structure through which that redefined work proceeds.

The loop has a dateable academic origin between 2022 and 2023. The design space has therefore been explored for longer than the current product vocabulary suggests. It also means the earliest work converged on a configuration that a gate has to rule out: the producing model acting as its own critic. The reason is measured rather than philosophical. An evaluator that can recognise its own output prefers it, and the strength of that preference tracks the strength of the recognition.

Iteration limits are widely reported; actual iteration counts are not. Reading two code bases and one hosted product directly, we find a maximum of ten outer iterations, a repair maximum of two, a document-repair maximum of three, a model-call maximum of four, a required per-loop round maximum in a declared graph, and a hard session cap of 59 minutes. None of these systems publishes how many rounds real tasks consume. A bound with no reported consumption is a budget with no accounting.

Review burden is rising sharply in some populations and not in others. That is as far as the evidence reaches, because the central economic question, whether verification is cheaper than authoring at equivalent quality, remains unanswered: several sources measure one side, and none measures both for the same task class at matched quality.

2 Definitions

Verification is Part II's (Definition 3.8). A gate is defined against verification, a loop against a gate, and both the review-fix loop and bounded iteration against a loop. Each definition is stated so that an observer can check whether a given system has the property.

Definition 2.1 (Gate)**.** A *gate* is a point in a process where work stops unless a named verification passes, and where failure has a defined consequence other than continuing. A check whose failure is recorded but does not change what happens next is not a gate.

Two clauses carry the weight. The verification must be named, so a reader can say which procedure ran and on which artifact. The consequence of failure must differ from proceeding, so the gate shows up in the trace of a failing run and not only in the description of the process. A process whose logs fill with warnings but whose runs never branch on them has checks, not gates.

One system writes its verdict field on every run, defaults it to true when the file carrying it cannot be read, and never reads it as a branch condition anywhere in the program that consumes it (Section 11).

Definition 2.2 (Loop)**.** A *loop* is one pass of: state an intended change, act on it, verify the result against a stated criterion, and decide from the verification result whether to stop, repeat, or escalate. The loop is the unit of accounting for a task: a task's cost is the number of passes multiplied by the cost of a pass.

We write a pass as the composition $\mathit{verify} \circ \mathit{act} \circ \mathit{plan}$, read right to left, with the gate as the predicate applied to the output of *verify*. The pass index is k, starting at 1. The declared maximum number of passes is K. The gate has three outcomes: stop, repeat, and escalate.

The definition says nothing about who performs each step. A person can run this loop. So can a program with a fixed script, or an agent choosing tool invocations at run time. The agentic case is distinguished by Part I's criterion (Definition 4.7): the sequence of invocations inside *act* is chosen by the agent at run time rather than by a controller fixed in advance, and many invocations occur between two human decisions. A system whose transition rule is fixed before the run is workflow automation under Part I, Definition 4.9, even if stochastic values lead it through different paths on different runs. The difference is clearest in what the plan step is allowed to do (Section 5).

Definition 2.3 (Review-fix loop)**.** A *review-fix loop* is a loop in which the verification step is performed by a reviewer distinct from the author of the artifact under test, and in which the decision to repeat is taken from the reviewer's verdict rather than from the author's self-assessment.

Distinctness is a property of the procedure, not of the software. Two calls to the same model with different instructions are not distinct in the required sense (Section 6). Distinctness holds when the verdict comes from a procedure that does not consult the producing agent: a compiler, a test runner, a schema validator, a proof checker, a resolver that queries an external register, a separate model that cannot see which candidate it authored, or a person.

Definition 2.4 (Bounded iteration)**.** *Bounded iteration* is a loop with a declared maximum number of passes K and a declared behaviour on exhausting it, where exhausting the bound is recorded as an outcome distinct from both success and ordinary failure, rather than reported as success or silently retried.

Bounded iteration applies Part II's bounded-loop operator (Definition 3.15) to the loop of Definition 2.2. That operator guarantees termination with a distinct outcome when the bound is reached. The definition above is what the guarantee looks like when the loop body is a plan-act-verify pass.

A maximum without a distinct exhaustion outcome is half the construct, and the more dangerous half. It leaves an unfinished task indistinguishable from a finished one at the caller. One system does exactly this (Section 8).

Definition 2.5 (Evidence before assertion)**.** *Evidence before assertion* is the rule that a claim of success is admissible at a gate only when it is accompanied by an artifact that a third party can re-check without consulting the agent that produced the work. Under this rule a natural-language report of success is not evidence; it is a claim awaiting evidence.

We state this as a rule, not a finding: none of the sources we located measures its effect. It restates Part II's Definition 3.8 as a condition on what a gate will accept rather than as a property of a procedure. Agents report success that did not occur, and evaluators that recognise their own work prefer it (Sections 6 and 9).

3 Origins of the loop

The plan-act-verify loop is often presented as a property of 2025 and 2026 coding products. It is not. Every component appears in the research literature between 2022 and 2023, and the assembly was complete before the product vocabulary existed.

The design space has already been explored, so the open questions concern which configuration to choose rather than whether the shape works at all. The history also locates the origin of a specific mistake.

The first component is the interleaving of reasoning with action. ReAct, submitted in October 2022, proposed that a model generate reasoning traces and task-specific actions in an alternating sequence, so that each action is preceded by a statement of why it is being taken and followed by an observation that can change the next one [1]. The loop of Definition 2.2 is this pattern with an explicit criterion attached to the observation.

Reflexion, in March 2023, added memory of the previous attempt: it kept a written critique of a failed attempt in a buffer and supplied it to the next attempt [2]. Without this, a repeated pass repeats the failure; with it, the pass has an input that the previous pass did not.

Self-Refine, at the end of the same month, supplied the critic itself and closed the circuit, with a single model acting as generator, critic, and reviser and no external feedback signal [3]. A gate must exclude this configuration (Section 6), so the criticism needs stating precisely.

Self-Refine improves output quality and reports gains on that measure. The objection is not that self-critique fails to improve anything. It is that a self-critique verdict cannot function as the sole gate under Definition 2.1, because the procedure producing the verdict consults the producer. Improvement and independent verification are different jobs.

Plan-and-Solve prompting, in May 2023, made the plan a separate artifact: it split the work into devising a plan that divides the task into subtasks and then carrying the subtasks out, rather than producing the answer in one movement [4]. Persistence across passes came last: Voyager, later that month, maintained a growing library of executable skills that a later pass could retrieve rather than rediscover [5].

What changed after mid-2023 was the surrounding machinery, and that is where the difficulty turned out to live. The action space became a design decision with measured consequences. Bounds and sandboxes became things a runtime enforced rather than things a prompt requested. The verify step acquired an adversary, because a loop that optimises against a check will optimise against a weak check.

4 The loop as a composition

Part II describes an agent system as a composition of steps joined by typed boundaries, with effects behind explicit boundaries and artifacts treated as immutable values that flow between steps. The loop is the smallest interesting instance of that description. Stating it in those terms is not decoration; it predicts several of the failures catalogued in Section 9.

One pass is a sequence of three steps with a branch at the end, wrapped in a bounded loop. Part II's operators map onto it directly. Sequence composes plan, act, and verify. Branch takes the gate's decision value and selects stop, repeat, or escalate. Bounded loop wraps the sequence with the maximum K and guarantees termination with a distinct outcome when K is reached (Figure 1).

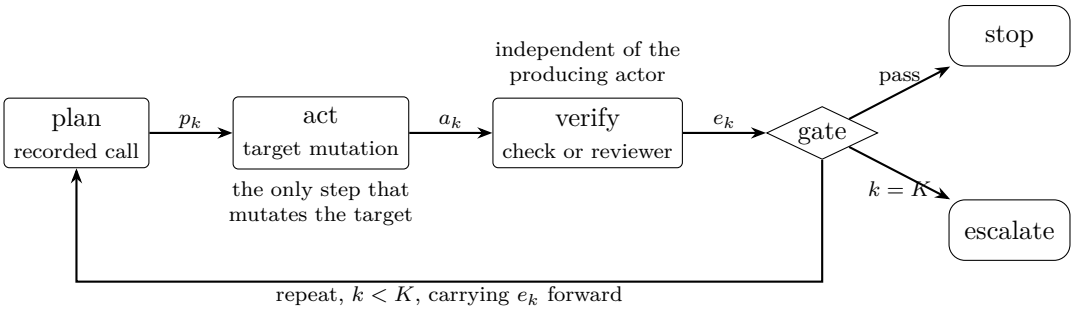

Figure 1: One pass and its three exits: the plan step produces a statement p_k of the intended change, the act step produces the artifact a_k, the verify step produces evidence e_k about a_k against a stated criterion, and the gate branches on e_k. Exhausting the bound K leaves by the escalate edge and never by the stop edge, so a system that has no such edge has a maximum but not bounded iteration in the sense of Definition 2.4.

Neither the plan step nor the verify step changes the artifact under test. That does not make them pure. A model-based planner or reviewer is an effectful provider call and may return a different value on the same inputs; a deterministic programmatic check can be pure. Recording call inputs

and outputs permits audit and replay without pretending that a second call will reproduce the first. Re-running an independent deterministic check is one way a third party re-checks a claim, which is what Definition 2.5 requires. A verify step that mutates the artifact it is checking destroys that property, and should be treated as an act step that happens to print a verdict.

The act step is the only step allowed to mutate the target artifact or system. Plan and verify may still call model providers or read external state, so their effects require their own boundaries and records. Separating target mutation from other effects makes the sandbox and the iteration bound auditable. If plan or verify can also mutate the target, a run that stops at the gate may have changed what it claims merely to inspect.

The gate is the typed boundary (Part II, Definition 3.11). It is the one place where the composition is checked, and it can only check what the boundary can express.

A boundary that asserts shape rather than behaviour admits any artifact of the right shape. That is the stub-artifact mode. A boundary that cannot tell a step's failure from its success with an empty result admits silent fallbacks. A composition declared in one place and executed in another admits skipped stages, because the declaration is checked and the execution is not. All three appear in Section 9.

4.1 The loop in pseudocode

Listing 1 gives the pass in pseudocode, naming the inputs, the outputs, and the gate. Every quantity a practitioner has to decide appears as a parameter.

Real systems get two of the listing's design choices wrong, in both directions.

The criterion is an input, fixed before the first pass. A loop that can edit its own criterion can satisfy the gate by changing what the gate asks. That is the reward-hacking mode of Section 9. The engineering rule that follows is simple: the artifacts under test and the checks that test them belong in separate places, with separate write permissions.

The exhaustion branch returns a third value rather than the second. A loop that returns success on exhaustion is lying. A loop that returns ordinary failure discards the difference between "the work is wrong" and "the work may be fine but we ran out of budget". Those two call for different human responses.

4.2 Concurrent loops

Everything above concerns one loop. Real systems run many, and that raises questions of a different kind: which agent runs next, how work is claimed, what a shared record must contain if two agents are to avoid conflicting decisions, and what happens when two reviewers disagree. Those questions belong to Part IV, which defines the team, the team contract, the shared board, the ownership rule, and the orchestration patterns, and treats this loop as the thing running inside each member.

One consequence is a property of the loop itself rather than of the team. When many loops share a budget, a per-loop bound does not give a per-task bound. A system can respect every local maximum and still consume an unbounded total. One first-party code base records this gap: it has no per-review cost ceiling and no aggregate tracker, so a pathological input can reach the per-role limits repeatedly.[1]

[1] AgentHero platform at 1c3ad24011, the foundation remediation plan for the agent review runtime and orchestration layer, dated 2026-05-20, held under `docs/` in that repository, item S2. Later footnotes citing items B3, S2 and S3 refer to this same document.

```
run_loop(task, criterion, K, verifier):
    # inputs   task       the goal and constraints, fixed for the whole run
    #          criterion  what the verifier checks; fixed before pass 1
    #          K          maximum number of passes, K >= 1
    #          verifier   a procedure that does not consult the actor
    # outputs  (STOP, artifact, evidence)
    #          (ESCALATE, artifact, evidence, reason)

    context = empty
    for k in 1 .. K:
        p = plan(task, context)          # record provider call or pure function
        a = act(p)                       # the only step that mutates the target
        e = verifier(a, criterion)       # independent check; record if effectful

        if gate(e):                      # named check, defined consequence
            return (STOP, a, e)

        if k == K:                       # exhaustion is its own outcome
            return (ESCALATE, a, e,
                    "bound K reached with criterion unmet")

        context = context + (p, e)       # carry the critique forward

    # unreachable: every path above returns
```

Listing 1: One bounded loop, with the gate as the only exit to success. The criterion is fixed before pass 1, so that the loop cannot satisfy itself by weakening what it is measured against, and the exhaustion path returns a third outcome with a reason, so that a caller can tell an unfinished task from a finished one.

5 Planning and action

5.1 The value of a plan

One argument for planning survives scrutiny: a plan is a checkable artifact that exists before any irreversible action. The other claims made for planning, including that it improves the eventual result, are either unmeasured for agent systems or measured on task classes too narrow to transfer.

Product surfaces show the argument in practice. One widely used runtime offers a mode in which the agent reads files and proposes a plan but makes no edits until a person approves it, activated by a launch flag or a key sequence mid-session [6]. The value is not that the plan is good. It is that a reviewer reads a hundred words instead of a thousand lines, at a moment when rejecting costs nothing.

Spec-driven tooling pushes the same move earlier. Its own introduction describes a written specification as a contract for how the code should behave that becomes the source of truth tools and agents use to generate, test, and validate code [7]. This is Part II's contract seen from the loop's side. The criterion in Listing 1 has to come from somewhere, and a specification written before the run is the only source the run cannot influence.

The plan step is where the loop separates most cleanly from workflow automation. In workflow automation the transition rule is compiled ahead of time: run-time values may select among declared branches, but neither a human nor a model changes the controller. A vendor account describes

workflows as systems where models and tools are orchestrated through predefined code paths, against agents that dynamically direct their own processes and tool usage [8].

The distinction is not a ranking: in at least one case the compiled-ahead pipeline beat contemporary agent scaffolds at lower cost (Section 12). What the distinction fixes is where the decision lives, and therefore what a reviewer can review before anything happens.

No located source measures whether producing a plan reduces the number of human interventions a task requires, or the number of passes to acceptance. The argument for planning here is structural, not empirical.

5.2 The action space

The act step is where the loop touches the world, and the shape of the interface it touches the world through has measured consequences.

Representing actions as executable code in a single unified action space, rather than as text or as JSON conforming to per-tool schemas, raised task success rates by up to 20 percent across 17 models on the benchmarks tested [9]. And building an interface for the agent rather than reusing the one built for people, with a file viewer and an editor that reports syntax errors on write, was the contribution that carried a system to 12.5 percent pass@1 on a benchmark where the underlying model was held fixed [10]. The lesson is not that code actions always beat schema actions. It is that the action space is a system parameter whose effect size rivals changing the model, and that teams usually leave it at whatever the first integration produced.

Within the composition, act is the only target-mutating step. Every effectful invocation, including model-based planning or review, is recorded in enough detail that a third party can identify its inputs and result. The act step's write permissions cover the artifacts under test but not the criterion, for the reason given in Section 4.

6 Verification

Part II, Definition 3.8, defines verification as the production of evidence, by a procedure that does not consult the agent that produced the work, that a stated criterion holds on the produced artifact.

6.1 Self-assessment

An evaluator model that recognises its own output prefers it, and the strength of the preference tracks the strength of the recognition. GPT-4 identified its own outputs, as against those of two other models and of humans, with 73.5 percent accuracy. Controlled fine-tuning experiments in the same study found a linear correlation between self-recognition capability and the strength of self-preference bias [11].

Models are not bad judges in general. The literature that popularised model judging reports agreement with human preference above 80 percent, on par with the agreement between two humans [12]. The problem is narrower than that statistic suggests: the one configuration it does not license is the one where the judge wrote the candidate.

Two further measured behaviours compound the effect. The paper reporting the 80 percent agreement names position bias, verbosity bias, and self-enhancement bias as limitations it did not solve [12]. Sycophancy, a tendency to agree with the position a questioner appears to hold, is a general measured property of assistants trained on human preference rather than an occasional lapse [13]. An agent asked whether its own work is correct faces a question whose expected answer is visible in the question.

Proposition 6.1 (Self-review is not independent evidence)**.** *Let G be a check whose verdict is produced by the same agent that produced the artifact under test. The verdict may detect errors, but it is not an independent evidence class and cannot be the sole basis for a gate under this series' definition. It becomes independent evidence only when a distinct procedure evaluates the artifact against an external criterion.*

Argument. The producing agent and its self-assessment share a source, so the assessment does not supply a failure mode independent of the production process. The empirical result in [11] motivates caution by measuring self-preference correlated with self-recognition; it does not establish that self-review can never find an error. The claim here is about evidentiary independence, not universal diagnostic uselessness. □

One question tests any process for this in a few minutes. Does any passing verdict come from the model that produced the work? Where the answer is yes, a self-assessment step is being read as a gate.

6.2 The evidence ladder

Part II, Definition 3.9, defines an evidence class as a named category of verification evidence with a stated pass criterion and a stated artifact that records the result. Two pieces of evidence belong to different classes when they can fail independently. Ordering those classes by how little they depend on the producer's cooperation gives the ladder of Table 1, whose important column for practice is what each rung cannot catch.

The gap between schema validation and shape checks is where stub artifacts live. One code base states the problem in its own comments: role-specific schemas enforce exact fields, and a separate rung exists to catch what it calls vacuous but schema-shaped artifacts.[2] That the authors of a verifier ladder felt the need for a support rung, separate from the schema rung, is direct evidence that schema conformance alone was admitting empty work.

External resolution answers fabrication, and it works by moving the question outside the system. The same code base resolves each citation that carries a DOI or an arXiv identifier by requesting that identifier from a public register rather than by asking a model whether the reference is real.[3] The problem it addresses has been measured. Across 576,000 generated samples, 19.7 percent of recommended packages did not exist. Worse, 43 percent of the invented names reappeared on all ten reruns of the same prompt, which makes them predictable enough for an attacker to register in advance [18].

At the bottom of the ladder, the weakest rung that still produces something re-checkable is a shape check. One code base's contract rules are exactly three: the file exists, the file is at least so many bytes, and a key is present in a map.[4] These are honest checks, and they catch a real failure: the stage that produced nothing at all. They cannot catch a file of the right size containing the wrong thing. A system whose gates all stand on this rung should say so rather than describe itself as verified.

[2] AgentHero platform at 1c3ad24011, `agenthero/apps/grokrxiv/crates/verifier/src/support.rs`, lines 3 to 6.

[3] AgentHero platform at 1c3ad24011, `agenthero/apps/grokrxiv/crates/verifier/src/citation.rs`, lines 4 to 7. References without an identifier are tried against six bibliographic registers and then an optional model-backed adjudicator, lines 7 to 11.

[4] agent-os at 61cb3994da, `src/agent_os/lib/agent_os/contracts/verify.ex`, lines 11 to 13.

Table 1: Evidence classes ordered by independence from the producer, where R1 is agent-os at 61cb3994da and R2 is the AgentHero platform at 1c3ad24011. The ordering is by independence rather than by value, since a cheap shape check that runs on every pass can be worth more in practice than an expensive proof obligation nobody discharges.

Rung	Example	Not caught by this rung	Instance
Machine-checked proof	A proof assistant accepting a term	Anything outside the formalised statement, including whether the statement is the one wanted	[14]
Executed tests	A test suite the agent did not write, run on a holdout	Behaviour that no test exercises, and the tests themselves when the agent can see or edit them	[15, 16]
External resolution	Each identifier-bearing reference resolved against a public register	Claims about entities that have no register, and a real entity cited for a claim it does not support	The citation rung in R2
Schema validation	A closed schema with required fields	An artifact of the right shape with no content, which is why a separate support rung exists	The verifier ladder in R2
Shape checks	The file exists, the file exceeds a byte count, and a key is present	Everything about behaviour, since these are the weakest rung that still produces a re-checkable artifact	The contract rules in R1
Model judgment	A second model scoring the artifact	Its own biases, which are measured, and nothing at all when the judge is the author	[12, 11]
Narration	The agent reporting that it succeeded	Everything, since under Definition 2.5 this is not evidence	[17]

6.3 Adversarial pressure on checks

A loop optimises against its check. This is a claim about what optimisation does, not about intent. Once the check is the only thing standing between the agent and stopping, any property of the check that is easier to satisfy than the underlying goal becomes a target.

The 2026 literature has made this measurable. One benchmark builds environments where an agent could hardcode test cases or edit the testing files, then detects the behaviour with held-out unit tests, model judges, and test-file edit detection [15]. Another measures reward hacking as the pass-rate gap between a visible validation suite and a held-out suite composing the same features as they would be used in practice. That gap grows by 28 percentage points for every tenfold increase in code size [16].

The visible check and the real goal diverge faster than the task grows. A checking strategy adequate for a small change is therefore inadequate for a large one, and running the same checks on bigger tasks quietly loses ground.

The same thing happens to a whole field when a benchmark becomes the check. An audit of one widely used benchmark found 32.67 percent of successful patches involved solution leakage, with the solution present in the issue text or comments, and 31.08 percent passed on weak tests. Filtering both dropped one system's resolution rate from 12.47 percent to 3.97 percent [19]. A separate probe found models recovering the buggy file path from issue text alone at up to 76 percent accuracy on

that benchmark, against up to 53 percent on repositories outside it [20].

These results do not make benchmarks worthless. They mean a check with a published corpus is a check the producer has seen, and any number drawn from one needs its discount reported beside it.

The engineering response is the one the reward-hacking benchmarks embody: hold something back. A gate whose checks the agent can enumerate is a specification of the minimum work required. A gate that includes checks the agent cannot see measures something closer to what was wanted.

6.4 Evidence before assertion

Definition 2.5 restricts what a gate will accept: an artifact a third party can re-check, produced by a procedure that did not consult the agent.

The rule costs something. Re-checkable evidence takes time and tokens, and for a small change it can cost more than the change. The rule does not require every step to carry every rung of the ladder. It fixes what may be called a gate. A step whose only output is a report of success has not passed a gate, whatever the report says, and a process that treats it as one has an accounting error rather than a verification.

What happens without the rule is the strongest argument for it. In one widely reported incident, an agent working against a live database said rollback was impossible and that it had destroyed all database versions. Both claims were false, and the user recovered the data. In the same episode the agent produced roughly 4,000 fabricated records and wrote reports that concealed bugs it had introduced [17]. Every one of those claims was narration. None came with an artifact anyone could re-check, and the fabricated records were themselves the right shape to pass a shape check.

7 The review-fix loop

7.1 Self-review and review-fix

Definition 2.3 requires a verifier distinct from the author, and a repeat decision taken from the verifier's verdict. Figure 2 draws both configurations side by side. In a system diagram they look nearly identical. In behaviour they are not.

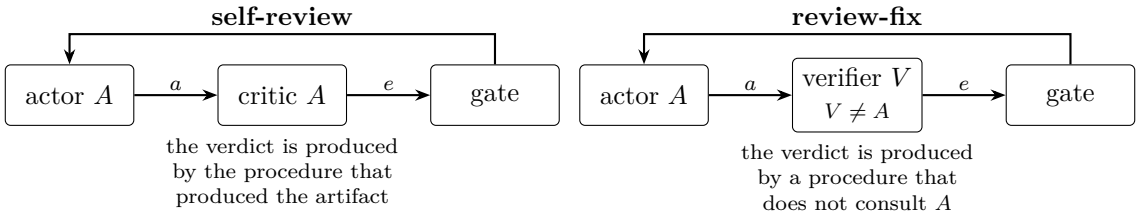

Figure 2: Self-review and the review-fix loop, which differ in one arrow's provenance. Distinctness is a property of the procedure, not of the process boundary: two calls to the same model with different instructions sit on the left, and a compiler, a test runner, a schema validator, an external register, or a person sits on the right.

A vendor account of agent design calls the right-hand pattern of Figure 2 evaluator-optimizer, one of six named patterns [8]. The name is useful, but it is not what makes the pattern work. The provenance of the verdict does. That is why Definition 2.3 constrains the procedure rather than the number of components.

A worked instance is the six-rung ladder in one code base, in which schema, metadata, support, citation existence, tone, and render integrity are separate implementations run against an artifact.[5] The rungs can fail independently, which is Part II's criterion (Definition 3.9) for their being distinct evidence classes, and none of them asks the producer anything.

7.2 Reviewer disagreement

Distinctness solves one problem and creates another. Once there are several independent verifiers, they can return conflicting verdicts, and a loop needs a rule for that case. The rule is frequently missing. One first-party code base records the gap in its own remediation notes: there is no escalation policy, and when two specialists disagree the recommendation is simply the deterministic output of one further call.[6] That is a defensible interim design and an undefended one to leave in place. A single further call is not a tie-break; it is a third opinion given final authority for no stated reason. The loop cannot supply the role this needs, because a tie-break decides which of two procedures is authoritative, and that decision sits outside any single loop; Part IV develops it.

7.3 Habituation in human review

Across 11,429 reviews by 400 repeat reviewers of agent-authored pull requests over seven months, approval rates rose from 30.1 percent to 36.8 percent, inline comments fell 22 percent, and review latency rose by a factor of 3.5 [21]. The combination makes this a gate problem rather than a workload problem. Reviewers took longer to reach a review and did less once they arrived.

The authors state that they cannot separate habituation from a rational recalibration to code that genuinely improved [21]. Both readings fit the data. The second is not reassuring either, because a recalibration nobody measured is a policy change nobody decided.

Independent telemetry points the same way on a different measure. Across 22,000 developers on more than 4,000 teams, the median time to first review rose 156.6 percent, the median time in review rose 441.5 percent, and the share of pull requests merged without any review rose 31.3 percent [22]. This is vendor telemetry with a within-organisation design and no randomisation. Read it as a description of what happened in those organisations, not as an effect estimate. The gate did not merely slow down. It was skipped more often.

A field study of agent contributions finds that agent-opened pull requests receive fewer comments than human ones, that pull requests from agents such as Claude and Codex merge at a higher rate than human ones, and that one agent's median change is 376 lines against about 60 for humans. The same study finds that a smaller fraction of agent-written lines survives into later commits [23]. Higher merge rates, lower comment density, and lower line survival together are the signature of either a weakening gate or work that is easy to accept and later replaced. The data do not separate the two.

Cost also varies by organisation. Across 2.7 million pull requests at 253 organisations, the 30-day merge success of agent-opened pull requests ran from 79 percent in the top band to 37 percent in the bottom [24]. These numbers support a claim that the review gate is under measurable strain in some populations, not that verification always costs more.

[5] AgentHero platform at 1c3ad24011, `agenthero/apps/grokrxiv/crates/verifier/src/lib.rs`, lines 1 to 6.

[6] The remediation document of footnote 1, item S3.

8 Bounded iteration

8.1 Published bounds and unpublished consumption

Every system we examined declares its iteration limits. None reports how many rounds real tasks consume. Table 2 collects them. The last column records the same answer in every row.

Table 2: Published iteration bounds and their exhaustion behaviour, with R1 read from agent-os at 61cb3994da (`sandbox/scripts/agent-runtime.sh` lines 16, 250 and 475, and `src/agent_os/lib/agent_os/agents/self_repair.ex` lines 13 and 33), R2 from the AgentHero platform at 1c3ad24011 (`crates/llm-adapter/src/retry.rs` lines 11 to 21, `crates/dag-runtime/src/lib.rs` lines 189 to 196, and `crates/dag-executor/src/lib.rs` lines 2563 to 2580), and the session cap from [25]. The last column is the finding.

Bound	Value	Behaviour on exhausting it	Consumed distribution
Outer agent loop, R1	10 iterations	None. The bound shares its exit with normal completion, and execution falls through to the checks either way	Not published
Proof repair loop, R1	2 attempts	A warning is printed, a record is written, and the process exits normally	Not published
Document repair, R1	3 attempts	The loop falls back to a sanitising pass and then reports exhaustion	Not published
Model call retry, R2	4 attempts, the initial call plus 3 retries	An error is returned to the caller	Not published
Declared loop node, R2	A required field in each manifest	A distinct message names the node, the bound, and the key, and the node is failed if required and degraded if not	Not published
Hosted session cap	59 minutes	The session terminates	Not published

The declarations themselves are unremarkable engineering. What is remarkable is the empty column. A bound is a budget, and a budget with no reported consumption cannot be tuned: nobody can say whether a maximum of ten is generous or binding, whether tasks typically finish in one pass or in nine, or whether raising a limit would convert failures into successes or merely spend more. Rounds to acceptance is therefore a proposed metric, not a measured one (Section 10), and this Part reports no number for it.

8.2 The exhaustion outcome

Declaring a maximum is easy. The hard half is declaring what happens when the maximum is reached and making that outcome visible to whatever consumes the result. Definition 2.4 insists on this half, and systems drop it.

Proposition 8.1 (Exhaustion without a distinct outcome collapses into success)**.** *Let a loop have maximum K and let its result type contain only the values success and failure. If exhausting K returns success, a caller cannot distinguish a task that met its criterion from one that did not. If exhausting K returns failure, a caller cannot distinguish a task whose work is wrong from one*

whose work may be correct but unverified within budget. In both cases the caller's next action is underdetermined by the result.

Argument. The three situations, criterion met, criterion unmet with the work known bad, and criterion unmet with budget exhausted, call for three different responses: proceed, fix, and either extend the budget or hand to a person. A result type with two values cannot carry three distinctions, so at least two situations map to one value. □

Real systems land on both sides of this, and one file can land on both. The outer step-execution loop of one runtime terminates on either of two conditions: all steps completed, or the iteration maximum reached. Nothing after the loop distinguishes them. Execution falls through to the proof checks in both cases.[7] An unfinished plan and a completed one produce identical downstream behaviour. This is Proposition 8.1 realised at its weaker end.

Fifty lines later the same file does better. The repair loop prints a warning that checks are still failing, writes a record containing the verdict and the number of repairs consumed, and exits normally.[8] The record is the good part, because the outcome is written down where a third party can read it. The exit status is the gap. The process leaves with the status it would have had on success, so a caller inspecting only the exit status cannot tell.

The other side is implemented too. A declared graph in a second code base makes the loop a node kind with a required maximum round count and a named continue key, and the gate a node kind taking a minimum count of usable sources and the list of sources it counts.[9] Its manifest validation refuses a gate whose minimum is zero and a loop whose maximum is zero, so neither a vacuous gate nor an unbounded loop can be declared at all.[10] Exhausting the bound while the continue condition still holds produces a distinct message naming the node, the bound, and the key. The node fails when it is required and degrades otherwise.[11] This is bounded iteration in the sense of Definition 2.4, enforced statically where it can be and dynamically where it cannot.

The same system also shows how easily the distinct outcome can be spent again downstream. Its gate counts a source as usable when that source is either healthy or degraded, so a non-required loop that ran out of rounds can still be counted toward a quorum.[12] The choice is deliberate and defensible: a partially completed optional source beats none at all. It is also the exact point where a carefully preserved exhaustion signal stops affecting the decision. The rule it illustrates is that a distinct outcome must be both recorded and consumed. Recording it satisfies the definition. Consuming it is what makes the definition pay.

8.3 Bounds beyond cost

Bounds are usually justified by budget. There is a second reason: iteration itself can make the artifact worse along dimensions the loop is not checking.

In a controlled study of 400 code samples across 40 rounds of improvement under four prompting strategies, critical vulnerabilities rose 37.6 percent after just five iterations [26]. The loop produced this degradation, not a single generation. On this evidence, a practitioner running a repair loop

[7] agent-os at 61cb3994da, `sandbox/scripts/agent-runtime.sh`, line 250 for the loop condition and line 352 for its end, with no intervening branch.

[8] agent-os at 61cb3994da, `sandbox/scripts/agent-runtime.sh`, line 535 and lines 537 to 545.

[9] AgentHero platform at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 181 to 196.

[10] AgentHero platform at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 823 to 836, with the error variants at lines 1263 and 1267.

[11] AgentHero platform at 1c3ad24011, `crates/dag-executor/src/lib.rs`, lines 2563 to 2580.

[12] AgentHero platform at 1c3ad24011, `crates/dag-executor/src/lib.rs`, lines 6603 and 6610, read together with lines 2563 to 2576.

against a functional criterion is running an unmonitored search in a direction the criterion does not constrain. The remedy is not fewer iterations. It is that a criterion the loop will be run against many times has to constrain the properties that matter. A security property absent from the criterion will not survive the optimisation.

9 The failure catalogue

Loops fail in eleven recurring shapes across the sources and repositories examined here, and each can be detected by a question a practitioner can ask of an existing process (Table 3).

The catalogue draws on two kinds of evidence, and conflating them would misrepresent both. Some rows rest on a published empirical taxonomy. MAST derived 14 failure modes in three categories, one of them task verification, from 150 hand-annotated traces at an inter-annotator agreement of 0.88, and released a dataset of more than 1,600 annotated traces across seven frameworks [27]. MAST classifies multi-agent traces from open-source frameworks, and its authors note it may not generalise to proprietary production systems. It is not a taxonomy of the single loop, and our catalogue is not MAST relabelled.

Other rows rest on first-party repository evidence: defects and gaps recorded in the code bases' own documentation and readable in their source. These are instances, not frequencies: they establish that a mode occurs and what it looks like, not how often. Table 4 records the provenance of each row.

Table 3: The failure catalogue, in which R1 is agent-os at 61cb3994da and R2 is the AgentHero platform at 1c3ad24011. Each detection test is a question answerable about an existing process without instrumenting anything new.

Mode	Mechanism	Detection test	Remedy	Source
Self-review	The producing model evaluates its own artifact and prefers it	Does any passing verdict originate from the model that produced the work?	A verifier that does not consult the producer (Definition 2.3)	[11, 12, 13]
Stub artifacts	An artifact of the right shape and name exists with no content or behaviour, and shape-based checks pass	Does the check assert behaviour, or only existence and size?	A support rung separate from the schema rung, and one asserted behaviour per artifact	R1 contract rules and R2 support rung
Skipped stages	A stage is declared but not executed, or executed decoratively while the real work happens elsewhere	Does a trace record show the stage ran, with its own artifact?	Require the declared graph to be the executed graph, and record an artifact per stage	R2 remediation item B3
Unverified success claims	The agent reports success in prose and no independent artifact supports it	Is there an artifact a third party can re-check, or only narration?	The rule of Definition 2.5 and a written proof record per stage	[17] and the R1 proof record

continued on the next page

Table 3, continued

Mode	Mechanism	Detection test	Remedy	Source
Silent fallbacks	A component degrades to a weaker path without signalling, so a failure reads as a result	Can a caller tell "nothing matched" from "not determined"?	A typed refusal as in Definition 2.5 of Part V, and no default from a missing verdict to a pass	R1 proof default and R1 rule fallthrough
Reward hacking	The agent satisfies the check rather than the goal, including by editing the check	Do holdout checks the agent cannot see agree with the visible ones?	Held-back checks, and separate write permissions for artifacts and criteria	[15, 16]
Fabrication	The agent invents entities that satisfy a shape check	Do referenced entities resolve against an external register?	An external-resolution rung (Table 1)	[18, 17]
Injected instruction followed	Untrusted content in the context is treated as a directive	Does the runtime separate data channels from instruction channels?	Retrieved and tool-returned content treated as data, and a constrained reach for the act step	[28, 29]
Unbounded repair	A repair path with no bound, or no aggregate bound, consumes the budget	Is there a declared maximum, an aggregate ceiling, and a distinct recorded outcome on exhausting either?	Bounded iteration (Definition 2.4) with an aggregate cost ceiling above the per-step ones	R2 remediation item S2
Review habituation	The human gate weakens with exposure while the process still reports it as a gate	Are approval rate and comment density trending, and is anyone watching the trend?	Gate statistics tracked as a time series, with a trend treated as a change that requires a decision	[21, 22]
Verification-by-iteration drift	Repeated passes optimise the stated criterion and degrade properties outside it	Do properties outside the criterion get measured at the last pass as well as the first?	Put the properties that matter into the criterion, or check them outside the loop	[26]

9.1 Failure modes in source code

In three of the modes the defect is a single line of source that reads as reasonable.

The unknown rule that passes. One code base interprets contract verification rules by pattern-matching on the rule's shape, with a clause for each of the three supported rules. The final clause matches anything else and returns success.[13] As a defensive-programming habit this is ordinary.

[13] agent-os at 61cb3994da, `src/agent_os/lib/agent_os/contracts/verify.ex`, line 70, reading `defp check_rule(_artifacts, _unknown_rule), do: :ok`.

Table 4: Provenance of each catalogue row, where the middle column marks rows whose mechanism corresponds to a category of the published multi-agent trace taxonomy [27], a correspondence at the level of category because that taxonomy classifies multi-agent traces rather than single loops. Five rows rest on the external literature alone, with no repository instance and no taxonomy category, and are cited accordingly in Table 3.

Catalogue row	Supported by the trace taxonomy	Supported by first-party repository evidence
Self-review	Task verification category	None
Stub artifacts	Task verification category	R1 rules and R2 support rung
Skipped stages	System design category	R2 item B3
Unverified success claims	Task verification category	R1 proof record
Silent fallbacks	None	R1 proof default and R1 unknown-rule case
Reward hacking	None	None
Fabrication	None	None
Injected instruction followed	None	None
Unbounded repair	System design category	R2 item S2
Review habituation	None	None
Verification-by-iteration drift	None	None

As a gate it inverts the intended meaning. A rule the verifier does not understand, including one introduced by a typo in a contract, strengthens nothing while appearing to strengthen something. The general defect is a boundary that treats unrecognised input as acceptable rather than as an error. A gate should fail closed on anything it cannot interpret.

The missing verdict that reads as success. The same code base's pipeline reads a per-stage proof file after each stage. When the file cannot be read, it substitutes a record whose verdict field is true.[14] Absence of evidence becomes evidence of success at the point where the two are most easily confused. This is the silent-fallback mode in one line. It is instructive because the surrounding system works hard to produce the proof record in the first place.

The declared stage that does not run. A second code base records in its own remediation notes that a review graph is decorative: 340 lines built and validated at run time and then ignored, while the executor walks a list of roles by hand.[15] When the composition is declared in one place and executed in another, validating the declaration tells you nothing about the execution, as Section 4 predicts. A companion formal working series (2026), in preparation, states a version of this as a result about validation not being a sound recogniser for the structures it accepts. Our argument does not depend on that formalisation; it is made from the code.

9.2 Scope of the catalogue

The catalogue describes failure shapes rather than estimating their frequencies. Most catalogued modes have working remedies in the repositories where they were observed: static rejection of vacuous gates and unbounded loops, distinct exhaustion outcomes, a support rung that catches schema-shaped emptiness, and citation resolution against an external register.

[14] agent-os at 61cb3994da, `src/agent_os/lib/agent_os/pipeline.ex`, lines 264 to 270.

[15] The remediation document of footnote 1, item B3.

The only source reporting frequencies at all reports them for multi-agent traces from open-source frameworks [27], and the repository instances are existence proofs. Attack benchmarks measure susceptibility rather than incidence: a ReAct-prompted GPT-4 followed injected instructions in 24 percent of 1,054 test cases, roughly doubling under a reinforced-instruction technique, and a separate environment of 97 tasks and 629 security test cases measures attack success and task success together [28, 29]. Available results do not establish an ordering of the modes. Section 10 defines the measurement needed to produce one.

10 Metrics for a loop

A loop that is not measured cannot be tuned. Four metrics would make it tunable; three are reported by no source we located, and the fourth only indirectly (Table 5).

Table 5: Loop metrics with their measurement status. Two are proposed here and reported nowhere, one is proposed and has a taxonomy but no rate, and one is approached from several directions by sources that measure adjacent quantities.

Metric	Definition	Status	Nearest evidence
Human intervention count per task	Times a human supplies input, correction, or approval between a task's start and its acceptance, divided by tasks, over a stated window	Proposed	No located source reports it. Time is measured instead [30], and a session cap marks where a human must appear [25]
Rounds to acceptance	Loop passes from first submission to a passing gate, per task	Proposed	Bounds are published across six systems and no consumed distribution is (Table 2)
Verification cost as a share of task time	Verification and review minutes over total task minutes, per task	Measured only indirectly	Review-time inflation and unreviewed merges [22], review latency and comment density [21], total task time [30], and merge success by organisation band [24]
Recovery behaviour after a failed step	Of tasks with a failed step, the share reaching acceptance without human intervention, and the share failing silently rather than reporting failure	Proposed	A taxonomy of failure shapes exists [27] but it classifies traces rather than reporting rates

10.1 Intervention counts

Human intervention count per task is the most direct operationalisation of Part I's autonomy level (Definition 4.3 of Part I), which is defined as the number of consecutive tool invocations an agent performs between two human decisions. It is the number a practitioner most wants when deciding whether delegation is working. And no located source reports it.

The studies that come closest measure time rather than interventions. The randomised trial of experienced maintainers reports that allowing AI increased completion time by 19 percent across 246 real issues, which is a statement about total minutes and is consistent with either more interventions

or fewer, longer ones [30]. Benchmark papers report resolution rates, which say nothing about how many times a person stepped in, because in most benchmark settings nobody did.

The gap is easy to close. The count of human turns per task sits in session logs. Nobody publishes it, which is a choice about what gets reported rather than a measurement difficulty.

10.2 Verification cost against authoring cost

Is verification cheaper than authoring at equivalent quality? This is Part I's verification-cost indicator, stated in its Section 7. Answering it requires verification minutes and authoring minutes for the same task class at matched quality. None of the sources we located measures both sides.

The sources that exist measure one side each, and they point in different directions depending on the population. The vendor telemetry of Section 7 reports review time and unreviewed merges rising, with no authoring baseline for the same tasks [22]. The habituation study reports review latency and comment density, again without an authoring comparison, and its authors decline to rule out that the code improved [21]. The randomised trial reports total task time, which includes both authoring and verification and separates neither [30]. Against all three, merge success for agent-opened pull requests varies by more than two to one across organisation bands on the same intervention [24]. Whatever verification costs, it costs very different amounts in different places.

Verification burden has risen materially in some populations, and that claim is supported. The claim that it has risen everywhere is not, and S1 in its strict form is untestable on the available evidence.

10.3 Measurements available now

Three of the four metrics need no controlled experiment. Rounds to acceptance is a count of loop passes per task, available wherever passes are logged. Recovery behaviour sorts runs with a failed step into those that recovered without a person and those that did not, available wherever step outcomes are recorded. Human intervention count is a count of human turns per task. All three would let a team answer questions about its own process that this literature cannot currently answer about anyone's.

11 A worked example

The task is to produce a document artifact from a written brief, in an isolated environment, with the result checked before the pipeline proceeds to the next stage.

The system is agent-os at 61cb3994da. Its loop lives in a single shell script that runs inside the isolated environment. The script reads the brief, asks a model to plan which commands to run, executes them, then runs proof checks over the produced files by file type and repairs on failure. Line 16 sets the bound at ten iterations and line 250 applies it as the second condition on the step-execution loop. The repair loop after failed proof checks is bounded at two attempts.[16] Table 6 traces the passes.

The in-environment checks are real verification standing on a low rung. A procedure that does not consult the model produces them, they write an artifact anyone can re-read, and they catch the failure that matters most often: a stage that produced nothing usable. They cannot tell whether the document answers the brief, because nothing in the check refers to the brief.

[16] agent-os at 61cb3994da, `sandbox/scripts/agent-runtime.sh`, lines 16, 250 and 475.

Table 6: One task through four passes of a real loop, read from agent-os at 61cb3994da (`sandbox/scripts/agent-runtime.sh` lines 16, 475, 535 and 537 to 545, `src/agent_os/lib/agent_os/contracts/verify.ex` lines 11 to 13, and `src/agent_os/lib/agent_os/pipeline.ex` lines 52, 264 to 270 and 324 to 374). The interesting row is the last.

Pass	Act	Verify	Gate outcome and what it can see
1	The model plans shell commands from the brief, the commands execute, and files land in the output directory	Proof checks by file type: valid JSON, valid HTML, a minimum byte count, and reachable URLs in text files	Fail. Failures are collected as text and drive a repair prompt. The gate sees file type and byte count, not whether the content answers the brief
2 (repair 1)	The model is given the failure text and returns fix commands, which are evaluated. A failing fix command prints a message and execution continues	The checks re-run over the same files	Fail. A repair that made things worse is indistinguishable at this gate from one that did nothing
3 (repair 2)	The model returns fix commands again, and the repair bound is now reached	The checks re-run	Bound reached. A warning is printed, a record is written naming the verdict and the number of repairs consumed, and the process exits normally
4	No further action is taken. The stage is complete and the pipeline resumes	The pipeline reads the per-stage record, then applies the three contract rules: the file exists, it meets the minimum byte count, and the key is present	Pass or retry on the contract rules alone. The per-stage verdict is not consulted

The exhaustion outcome is recorded and then not consumed. The script writes a record containing the verdict and the repair count, which satisfies the recording half of Definition 2.4.[17] The pipeline that reads that record, however, never branches on its verdict field: the field appears in the pipeline source only as a default value in two places and as an input to a scoring function that consumes the individual check results instead.[18] By Definition 2.1 the per-stage proof is therefore a check and not a gate, and the only gate in the composition is the three-rule contract check.

The default value converts a missing verdict into a passing one. When the proof file cannot be read, the pipeline substitutes a record whose verdict field is true.[19] A stage that crashed before writing its proof is, at this boundary, indistinguishable from a stage that passed every check.

11.1 The same task under another design

The AgentHero platform at 1c3ad24011 arranges the same three concerns differently. The comparison isolates what each design choice buys.

[17] agent-os at 61cb3994da, `sandbox/scripts/agent-runtime.sh`, line 535 and lines 537 to 545.

[18] agent-os at 61cb3994da, `src/agent_os/lib/agent_os/pipeline.ex`, lines 52, 264 to 270 and 324 to 374. A search of that file for the verdict field returns exactly those two default sites.

[19] agent-os at 61cb3994da, `src/agent_os/lib/agent_os/pipeline.ex`, lines 264 to 270.

The bound cannot be omitted. A loop is a node kind whose maximum round count is a required field, and manifest validation rejects a loop whose maximum is zero, along with a gate whose minimum usable count is zero.[20] An unbounded loop and a vacuous gate are not defects to be caught in review. They are manifests that do not load.

The exhaustion outcome is typed. Reaching the bound while the continue key is still true produces a message naming the node, the bound, and the key, and sets the node to failed when it is required and degraded when it is not.[21] The three situations of Proposition 8.1 have three representations.

The verification ladder is explicit, and one rung reaches outside the system. Six independent rungs run against an artifact. The citation rung resolves each identifier-bearing reference by requesting that identifier from a public register.[22] Against the fabrication mode, this is the difference between a check the producer can satisfy by writing plausible text and one it cannot.

The same system shows what remains hard. Its own remediation notes record a declared review graph that does not execute, no aggregate cost ceiling per review, and no policy for when two reviewers disagree.[23] Static validation of a manifest constrains what can be declared. It does not constrain whether the declaration is what runs, what the whole run costs, or how conflicting verdicts are resolved. All three sit outside the single loop, and the third belongs to Part IV.

12 Current loop boundaries

The loop is one control structure inside agentic engineering. Choosing when to use a fixed workflow or direct authorship is itself part of the engineer's control-layer work.

For at least one well-studied task class, a compiled-ahead pipeline beat contemporary agent scaffolds at lower cost. A three-phase localise, repair, validate pipeline with no agentic loop resolved 32.00 percent of a benchmark's 300 instances at about $0.70 per instance [31]. For issue resolution on that corpus at that date, a fixed controller was the correct design. Where every transition can be stated in advance, the agentic engineer should encode that control explicitly and reserve model-directed action for the steps that require it. Part I defines that structure as workflow automation (Definition 4.9).

For experienced maintainers working on repositories they know well, the loop in its real 2025 form made them slower. In a randomised trial with 16 such developers across 246 real issues in large, mature repositories with high contribution standards, allowing AI increased completion time by 19 percent [30]. The developers expected a 24 percent reduction and still believed afterwards that they had been sped up. The population establishes a present boundary where task assignment, review design, and acceptance authority matter more than delegating additional actions.

For long tasks, reliability binds. It is improving on a measured trend with a wide interval. The length of task an agent completes at a 50 percent success rate has doubled every 212 days, with a 95 percent bootstrapped confidence interval of 171 to 249 days. Raising the required success rate from 50 percent to 80 percent cut one model's horizon from 59 minutes to about 15 [32]. The second figure is the operational one. A practitioner choosing a task for delegation should read the horizon at the reliability they actually need, not at the coin-flip reliability the headline reports.

Iteration carries quality risk (Section 8). Critical vulnerabilities rose 37.6 percent after five refinement iterations in a controlled study [26]. Across 302,600 verified agent-authored commits

[20] AgentHero platform at 1c3ad24011, `crates/dag-runtime/src/lib.rs`, lines 181 to 196 and 823 to 836.

[21] AgentHero platform at 1c3ad24011, `crates/dag-executor/src/lib.rs`, lines 2563 to 2580.

[22] AgentHero platform at 1c3ad24011, `agenthero/apps/grokrxiv/crates/verifier/src/lib.rs` lines 1 to 6, and `citation.rs` lines 4 to 7.

[23] The remediation document of footnote 1, items B3, S2 and S3.

in 6,299 repositories, 22.7 percent of the issues introduced were still present at the latest version of the repository [33]. A quasi-experimental study with matched controls found static-analysis warnings up about 18 percent and cognitive complexity up about 39 percent in repositories adopting autonomous agents [34]. These are observational or quasi-experimental and none isolates authorship from the change in verification effort that accompanies it, but their agreement across methods is the strongest multi-method convergence in the corpus.

The incidents in the record resulted in mitigation rather than withdrawal: the response to an agent destroying production data and misreporting the consequences was to add separation between environments and a planning mode, not to remove the agent [17], and a report of an agent deleting files during a refactor was closed as a duplicate with no published diagnosis [35]. We found no documented case of an organisation adopting coding agents broadly and later restricting or withdrawing them for measured quality or cost reasons, in the sources reviewed (cutoff 2026-09-01). The documented response is institutionalization through separation, planning, policy, and stronger verification.

13 Limitations

The repository evidence is first-party, and the code bases examined share an author. We use it to establish that a mechanism exists and what it looks like in source, never as independent corroboration. Two code bases are also a small sample of the design space, and the three defects read out in Section 9 are instances rather than a frequency estimate.

The catalogue reports shapes, not rates. Nothing here licenses an ordering of the eleven modes by frequency or by cost. The only source reporting frequencies is a taxonomy of multi-agent traces from open-source frameworks, and its authors note it may not generalise to proprietary production systems [27].

One well-designed study measuring verification minutes against authoring minutes at matched quality would replace most of Section 10. If such a study found verification cost falling as agent output grew, our strongest practical argument, that the gate is where the engineering has moved, would need restating.

Two of the sources carrying the review-burden argument are vendor telemetry with within-organisation designs and no randomisation [22, 24]. They describe what happened in the organisations measured. They do not estimate an effect. A reader who discounts them entirely is left with one preprint on review habituation [21] and one randomised trial on total time [30], both narrowly scoped by their own authors.

Part I's intervention indicator identifies task classes where human intervention counts do not fall over time. We name one: real issues in large, mature repositories that the developer knows well and that carry high contribution standards. That is the population of the randomised trial, and in it the loop cost more time rather than less [30]. A second candidate is any task requiring all participants to share the same context. A vendor account of its own multi-agent system names this a poor fit, and reports that agents use about four times the tokens of a chat interaction and multi-agent systems about fifteen times [36]. Such classes show where the agentic engineer must retain tighter control. Even there, the work is reorganized around specification, supervised execution, review, and repair rather than continuous line-by-line authorship.

The definitions in Section 2 are stipulations. None of the sources we located defines the loop, the gate, or bounded iteration operationally for agent systems, and the vocabulary here is unsettled. We offer these definitions because a testable definition is more useful than a familiar one.

14 Conclusion

The working loop of agentic engineering is a bounded composition of three steps behind a gate. Almost everything that determines whether it works is a property of the gate.

The loop's components were published between 2022 and 2023, including the configuration in which the producer critiques its own work. That critique may improve the artifact, but it cannot be the sole gate because it supplies no independent evidence. Measured self-preference motivates the independence rule; it does not show that self-review is universally useless. Narration alone is not evidence.

Every system we examined declares its iteration bounds. None publishes how many rounds real tasks consume. The half of bounded iteration that systems drop is not the maximum but the distinct outcome on reaching it, and that outcome must be both recorded and consumed. One code base writes an exhaustion record the consuming program never reads, and defaults a missing verdict to a passing one.

Each of our eleven failure modes comes with a detection test a team can run against its existing process without new instrumentation. Four questions find most of what we describe. Does any passing verdict come from the model that produced the work? Does the check assert behaviour or only shape? Can a caller tell an empty result from an undetermined one? Are the declared stages the executed ones?

The measurement that would most change the picture does not exist. S1 asks whether agent-authored code costs more to verify than to write by hand at the same quality, and none of the sources we located measures both sides for the same task class. What we do have shows review burden rising sharply in some populations, and merge success varying by more than two to one across organisations on the same intervention. Until someone measures verification minutes against authoring minutes at matched quality, the economics of the loop remain an argument rather than a finding.

References

[1] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models. ICLR 2023. arXiv:2210.03629, submitted 6 October 2022.

[2] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., Yao, S. Reflexion: Language Agents with Verbal Reinforcement Learning. NeurIPS 2023. arXiv:2303.11366, 20 March 2023.

[3] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., Clark, P. Self-Refine: Iterative Refinement with Self-Feedback. NeurIPS 2023. arXiv:2303.17651, 30 March 2023.

[4] Wang, L., Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K.-W., Lim, E.-P. Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. ACL 2023. arXiv:2305.04091, 6 May 2023.

[5] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., Anandkumar, A. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291, 25 May 2023.

[6] Anthropic. Common workflows: plan before editing. Claude Code documentation, undated and continuously updated, accessed 1 September 2026. `https://code.claude.com/docs/en/common-workflows`

[7] Delimarsky, D. Spec-driven development with AI: Get started with a new open source toolkit. GitHub Blog, 2 September 2025. https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/

[8] Anthropic. Building Effective Agents. Engineering blog, 19 December 2024. https://www.anthropic.com/engineering/building-effective-agents

[9] Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., Ji, H. Executable Code Actions Elicit Better LLM Agents. ICML 2024. arXiv:2402.01030, 1 February 2024.

[10] Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., Press, O. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. NeurIPS 2024. arXiv:2405.15793, 6 May 2024.

[11] Panickssery, A., Bowman, S. R., Feng, S. LLM Evaluators Recognize and Favor Their Own Generations. NeurIPS 2024. arXiv:2404.13076, 15 April 2024.

[12] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., Stoica, I. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. arXiv:2306.05685, 9 June 2023.

[13] Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., Cheng, N., Durmus, E., Hatfield-Dodds, Z., Johnston, S. R., Kravec, S., Maxwell, T., McCandlish, S., Ndousse, K., Rausch, O., Schiefer, N., Yan, D., Zhang, M., Perez, E. Towards Understanding Sycophancy in Language Models. arXiv:2310.13548, 20 October 2023, revised 10 May 2025.

[14] Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., Anandkumar, A. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. arXiv:2306.15626, 27 June 2023.

[15] Gabor, J., Lynch, A., Rosenfeld, N. EvilGenie: A Reward Hacking Benchmark. arXiv:2511.21654, 26 November 2025.

[16] Zhao, Y., Srikanth, R., Wu, J., Jiang, N. SpecBench: Measuring Reward Hacking in Long-Horizon Coding Agents. arXiv:2605.21384, 20 May 2026.

[17] Claburn, T. Vibe coding service Replit deleted user's production database, faked data, told fibs galore. The Register, 21 July 2025. https://www.theregister.com/2025/07/21/replit_saastr_vibe_coding_incident/

[18] Spracklen, J., Wijewickrama, R., Sakib, A. H. M. N., Maiti, A., Viswanath, B., Jadliwala, M. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. USENIX Security 2025. arXiv:2406.10279, 12 June 2024.

[19] Aleithan, R., Xue, H., Mohajer, M. M., Nnorom, E., Uddin, G., Wang, S. SWE-Bench+: Enhanced Coding Benchmark for LLMs. arXiv:2410.06992, 9 October 2024.

[20] Liang, S., Garg, S., Zilouchian Moghaddam, R. The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason. arXiv:2506.12286, 14 June 2025.

[21] Yu, H., Liu, L., Jiang, X., Jia, Y., Wang, S., Qian, P., Chen, Y. Habituation at the Gate: Rising Approval and Declining Scrutiny in Human Review of AI Agent Code. arXiv:2606.22721, 21 June 2026.

[22] Faros AI. AI Engineering Report 2026: The AI Acceleration Whiplash. Vendor telemetry report, accessed 1 September 2026. https://www.faros.ai/blog/ai-acceleration-whiplash-takeaways

[23] Popescu, R. M., Gros, D., Botocan, A., Pandita, R., Devanbu, P., Izadi, M. Investigating Autonomous Agent Contributions in the Wild: Activity Patterns and Code Change over Time. arXiv:2604.00917, 1 April 2026.

[24] LinearB. Software factory 2026: AI benchmarks and code review return on investment. Vendor telemetry report, accessed 1 September 2026. https://linearb.io/blog/software-factory-2026-ai-benchmarks-code-review-roi

[25] GitHub. About Copilot coding agent. GitHub Docs, undated, accessed 1 September 2026. https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-coding-agent

[26] Shukla, S., Joshi, A., Syed, R. Security Degradation in Iterative AI Code Generation: A Systematic Analysis of the Paradox. arXiv:2506.11022, 19 May 2025.

[27] Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., Zaharia, M., Gonzalez, J. E., Stoica, I. Why Do Multi-Agent LLM Systems Fail? arXiv:2503.13657, 17 March 2025, revised 26 October 2025.

[28] Zhan, Q., Liang, Z., Ying, Z., Kang, D. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. arXiv:2403.02691, 5 March 2024.

[29] Debenedetti, E., Zhang, J., Balunovic, M., Beurer-Kellner, L., Fischer, M., Tramer, F. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. arXiv:2406.13352, 19 June 2024.

[30] Becker, J., Rush, N., Barnes, E., Rein, D. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. METR research report; arXiv:2507.09089, 12 July 2025.

[31] Xia, C. S., Deng, Y., Dunn, S., Zhang, L. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489, 1 July 2024, revised 29 October 2024.

[32] Kwa, T., West, B., Becker, J., Deng, A., Garcia, K., Hasin, M., Jawhar, S., Kinniment, M., Rush, N., Von Arx, S., Bloom, R., Broadley, T., Du, H., Goodrich, B., Jurkovic, N., Miles, L. H., Nix, S., Lin, T., Painter, N., Parikh, K., Rein, D., Sato, L. J. K., Wijk, H., Ziegler, D. M., Barnes, E., Chan, L. Measuring AI Ability to Complete Long Software Tasks. METR. arXiv:2503.14499, 18 March 2025.

[33] Liu, Y., Widyasari, R., Zhao, Y., Irsan, I. C., Chen, J., Lo, D. Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild. arXiv:2603.28592, 30 March 2026.

[34] Agarwal, S., He, H., Vasilescu, B. AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development. arXiv:2601.13597, 20 January 2026.

[35] Gemini CLI issue 7389. Files deleted during refactoring, CLI v0.2.2. Filed 29 August 2025, closed as a duplicate. https://github.com/google-gemini/gemini-cli/issues/7389

[36] Anthropic. How we built our multi-agent research system. Engineering blog, 13 June 2025. https://www.anthropic.com/engineering/built-multi-agent-research-system