PROCONTRACT: DECOUPLING RESPONSIBILITY AND PROGRESS FROM THE HARNESS IN LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agents are increasingly used to turn user-assigned responsibility into artifacts over hours of execution. Yet even with capable underlying models, their reliability is undermined by two persistent failure modes: *responsibility dilution* and *progress misjudgment*. Both arise because long-running agents continuously rewrite their execution state, such as delegating subtasks, compacting context, restarting, and even modifying their harness, while the responsibility and its progress remain entangled with that state. We introduce PROCONTRACT, whose progress-contract kernel independently keeps responsibilities in a Contract outside the agent and tracks progress against that Contract through authorized judgments. The agent's harness is deliberately disentangled from the kernel, so changes in execution state cannot alter assigned responsibilities or interfere with progress judgment. This separation also makes self-improvement an ordinary responsibility: the agent can rewrite its own harness while the kernel preserves the task it was asked to carry out. With GPT-5.6 Terra, PROCONTRACT raises the mean pass rate from 72.35% to 77.24% over the same model's public mini-SWE-agent run. The gain comes from the bottom: the number of tasks below 10% halves. A version written by the agent itself produced a successor that ran all 200 tasks under the same kernel. With GPT-5.6 Sol, PROCONTRACT raises the mean from 69.89% to 75.25% over the model's entry on the official ProgramBench leaderboard. With GPT-5.2 on Terminal-Bench 2.0, released after that model's knowledge cutoff, it passes 65.7% of tasks, above all four public harnesses evaluated with GPT-5.2.

1 INTRODUCTION

Long-horizon agents are increasingly used to carry user-assigned responsibilities over hours of execution, turning them into artifacts such as reconstructed programs, revised repositories, and completed command-line jobs (Yang et al., 2026; Jimenez et al., 2024; Merrill et al., 2026). Yet even with capable underlying models, they often fail not because they cannot solve a task, but because they lose track of the responsibilities. A requirement drops out as the agent shifts attention; the agent treats a passing check as the end of the task; or it reports success that the environment does not support (Advani, 2026). We call these failures *responsibility dilution* and *progress misjudgment*: the agent loses track of what it still owes and what has actually been established. The model sets the ceiling; preserving the task sets the floor.

The cause is that the responsibility and its progress are entangled with the agent's execution state. The responsibility is what the agent owes: its requirements, deadline, and budget; progress is what has been accepted as meeting them. Most agent harnesses, including mini-SWE-agent, keep both in the same working context that the agent continually rewrites as it works (Figure 1a). That context is compacted, reshaped as the agent changes approach or delegates work, and may be lost on restart. As a result, a requirement can drop out and a remembered passing check can stand in for progress that was never established. Better memory does not solve this, nor do durable workflows that persist execution while trusting the worker (Temporal Technologies, 2026). The problem is not merely what the agent can recall, but what it is allowed to change: responsibilities should survive changes in execution state, and progress should be recorded only by an authorized judgment of the outcome.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

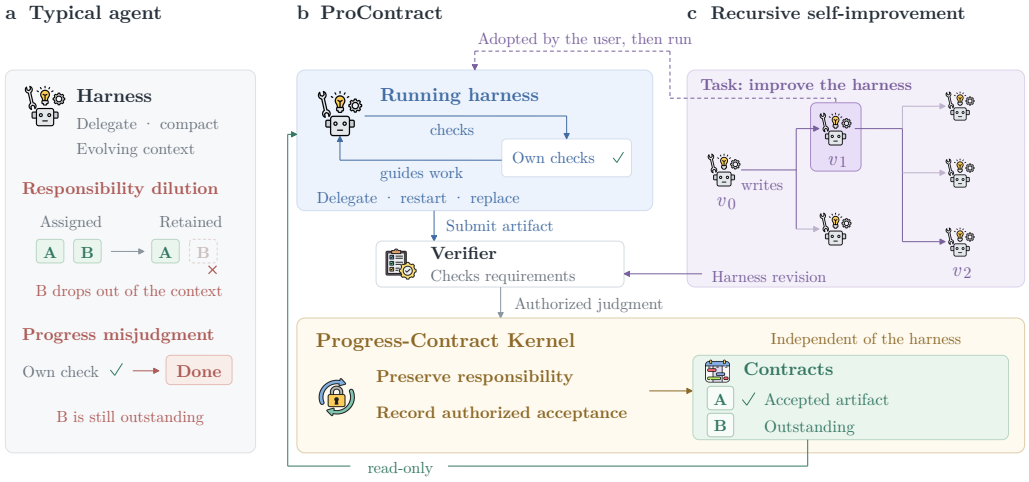

Figure 1: **ProContract architecture.** (a) In a typical agent, the responsibility and its progress live in the harness's growing context, so a requirement can drop out (responsibility dilution) and a passing own check can mark the task done (progress misjudgment). (b) ProContract keeps both in Contracts that only the kernel updates. The harness may delegate, restart, or be replaced, and it reads the Contracts only; its own checks guide work, and progress enters only as a verifier's judgment of a submitted artifact. (c) Improving the harness is one more task: v_0 writes v_1, which writes v_2 with its own code; each revision is judged like any other artifact.

PROCONTRACT addresses this entanglement by taking the responsibility and its progress out of the agent's execution state. A persistent *Contract* holds both, and only a *kernel* may update it (Figure 1b). The running harness reads the Contract and works in its own workspace; its own checks may guide execution but cannot record progress. To do so, the harness submits an exact artifact to a *verifier*, whose judgment the kernel records only if it is authorized and still matches the current requirements and that artifact. An acceptance therefore remains attached to what was actually judged: it survives a change of harness, does not transfer to a revised artifact, and is withdrawn if its verifier or the user retracts it, or if supporting acceptances are withdrawn. No harness can erase a responsibility or declare its own work completed. PROCONTRACT does not make the model more capable; it preserves what the model is capable of doing. With the same model, it passes 76% to 93% of the hidden tests on five ProgramBench tasks where the public mini-SWE-agent run scores zero.

Because the kernel preserves the responsibility and its progress, the agent's harness is free to change without changing the task. We disentangle the agent's *harness*, the instructions that decide what to try next and the code that builds its context, keeps its memory, and runs its tools, from the kernel. Long tasks need this flexibility: an approach may stall partway through, while harnesses are increasingly improved by automated methods and by agents themselves (Hu et al., 2025; Zhang et al., 2026; Lee et al., 2026). A successor reads the same Contract, so replacing the harness cannot by itself clear a responsibility or revoke an acceptance. A replacement may still interfere with the execution state, because a new harness may change how that state is stored. An external controller therefore installs a new *version* of the harness with its execution state migrated to it, and can roll back to the previous version and a matching checkpoint if the new one stalls.

Disentangling the harness from the kernel turns self-improvement into another task under the same kernel. When the agent's own harness becomes the object of its work, a revision it produces is treated like any other submission: it can be judged and recorded, but not automatically adopted. An external controller builds and qualifies the revision, a rule fixed in advance determines whether to continue exploration from it, and the user decides whether it becomes the next version. The process becomes recursive when a generated version runs its own code, including its instructions for producing successors, to create the next revision. Because evaluation and adoption remain outside what any version can modify, the agent can improve how it works without gaining the authority to redefine either its responsibility or when that responsibility is complete.

Our contributions are:

- **A progress-contract kernel that preserves responsibility and makes progress a judgment.** The kernel keeps the responsibility and its progress outside the agent's execution state and records progress only through authorized judgments on exact artifacts. On 200 ProgramBench tasks with the same model, PROCONTRACT increases the paired mean pass rate by 5.33 points and halves the number of tasks scoring below 10%.
- **A disentangled harness that lets the agent improve itself without losing the task.** It can be replaced mid-task while the Contract remains intact, making each new version another deliverable whose adoption is decided outside the agent. Across two rounds, a harness the agent generated wrote its own successor; the successor gained 6.37 points on six development tasks yet stayed unadopted, because exploring from a version is decided separately from adopting it, and it ran all 200 ProgramBench tasks under the same kernel with every Contract intact.
- **An agent that generalizes across models and benchmarks.** With GPT-5.6 Sol, PROCONTRACT improves by 5.36 points over the same model's entry on the official ProgramBench leaderboard and achieves a higher mean pass rate than all 21 official entries. On Terminal-Bench 2.0, released after GPT-5.2's knowledge cutoff, it passes 65.7% of tasks, the highest pass rate among harnesses evaluated with the same model on the public leaderboard.

2 PROBLEM SETTING: TASK EXECUTION AND PROGRESS MANAGEMENT IN LONG-HORIZON AGENTS

Long-horizon Language Model Agents. An agent pursues user-assigned *responsibilities* through iterative planning, tool execution, and progress tracking (Figure 1a). A *responsibility* specifies the requirements that an output *artifact* must satisfy, together with a deadline and a budget. Only the user may revise or release these responsibilities. The agent operates through a *harness* H, which defines its execution framework, including context and memory management and tool orchestration. A running instance of H has its own execution state and workspace. During long-horizon execution, the execution state evolves as the agent works toward its assigned responsibilities, while the harness itself may also be updated or replaced. To assess progress toward these responsibilities, we distinguish the agent's internal assessments from verified results. A *verifier*, such as a test suite, a metric threshold, a learned judge, or a human expert, evaluates a fixed artifact against the assigned requirements. A positive verification result, once validated and recorded, constitutes an *acceptance*. Since some requirements depend on the completion of others, acceptances may have dependencies. We define *progress* as the set of valid acceptances whose requirements and prerequisite acceptances remain satisfied.

The Problem: Responsibility Dilution and Progress Misjudgment. As tasks grow in complexity and execution length, agents often decompose user-assigned responsibilities into multiple subtasks. This allows agents to tackle complex tasks incrementally, but also requires them to track the requirements and progress of individual subtasks while maintaining the original responsibilities. As subtasks accumulate, agents increasingly rely on their evolving execution state to manage these records. This dependence creates two potential failure modes: *responsibility dilution*, where original requirements are lost, overlooked, or replaced by narrower subgoals without user authorization; and *progress misjudgment*, where agents incorrectly record progress without valid acceptance or fail to retain previously valid acceptances. The challenge becomes more pronounced during agent self-improvement. Beyond updating their execution state, agents may modify their execution logic or replace their own harness while the original task remains unfinished. When responsibility and progress tracking depend on the harness being modified, maintaining these records consistently becomes an additional challenge. These considerations motivate separating responsibility and progress management from the agent's execution state and harness, so that assigned responsibilities and verified progress can be maintained even as the agent's execution mechanism evolves.

3 PROCONTRACT

To address these two failure modes, PROCONTRACT separates the harness, which does the work, from the kernel, which preserves the responsibility and records only judged progress. Each responsibility lives in a *Contract* that only the *kernel* can change. The harness never records progress itself: it

submits an artifact, a verifier judges it against the requirements, for example by running tests or checking a metric, and the kernel records the judgment as an acceptance. The section develops this in two steps. First, the kernel preserves responsibility and admits progress only through valid acceptances, addressing both failure modes (Section 3.1). Second, because neither the responsibility nor progress lives in the harness, every harness operates under the same rules, the harness can therefore be replaced mid-task and even improved by the agent itself (Section 3.2).

3.1 The Progress-Contract Kernel

The kernel K is a deterministic program whose state $\mathcal{C}$ is the set of all Contracts. It runs no model; verifiers run outside it, and a running harness can request operations but cannot bypass them (Figure 1b). It has two jobs, corresponding to the two failure modes: preserving responsibility and admitting progress only through acceptances. The model decides how to explore, implement, and debug, but cannot complete a task by declaring it done. Only the user may change the responsibility; progress changes through two rules, *Accept* and *Withdraw*. We give the reference semantics (Appendix A).

Responsibility outside the harness. The first job is to keep the responsibility where no harness can reach it. A Contract stores the responsibility for one task τ: its requirements ρ_τ, which only the user can change, together with the deadline, the budget, and the resources used so far. Work is charged through the kernel, which rejects charges beyond the budget, while a controller outside the agent stops work at the deadline; in either case, the task stays outstanding. A delegated subtask gets a Contract of its own, and the parent can rely on it only through its acceptance, never through a subagent's report. A harness can still lose a requirement from its execution state, but it cannot remove or narrow it: the requirement stays in the Contract, every acceptance is judged against it, and the task stays outstanding until a submission is accepted against it. Even a perfect memory would not suffice, because the user can revise a requirement without the harness observing anything (Proposition 1).

Progress only by judgment. The second job is to record exactly what was judged: nothing unjudged counts as progress, and when a judgment is withdrawn, only the progress that relied on it is lost. A task may hold several acceptances because an acceptance settles the stated requirements, not the goal, and the harness may continue improving a finished task. Consider an agent that reconstructs a command-line tool and then packages it. Reconstruction is accepted first for an artifact x_0 and then for a faster x_1, which becomes the delivered version; packaging, which requires an accepted reconstruction, was accepted relying on one of the two. Now suppose the acceptance of x_0 is withdrawn because the reference used to judge it proves wrong. Reconstruction stays finished on x_1, and packaging must reopen only if its acceptance relied on x_0. A rule that sees only tasks, like the dependency links in an issue tracker, must either leave packaging accepted, incorrectly if it relied on x_0, or reopen it, unnecessarily if it relied on x_1. Tracking individual acceptances and their dependencies is therefore necessary (Theorem 1) and sufficient (Proposition 3) to withdraw every task whose delivered acceptance has lost its basis, and no others.

Progress is therefore recorded per acceptance. Each acceptance a binds the requirements under which it was judged, the frozen artifact, its evidence, the verifier's judgment, and the acceptances it relies on, $\mathrm{supp}(a)$. In addition to the responsibility, a Contract stores its task's acceptances; its current delivery σ_τ (empty while the task is outstanding); and a status z_τ that is `fulfilled`, `released`, or outstanding. An acceptance a is valid, written $\mathrm{Live}_{\mathcal{C}}(a)$, when four conditions hold: (i) it is not withdrawn; (ii) the requirements it was judged under are still current; (iii) it is locally sound, as defined below; and (iv) every acceptance it relies on is itself valid:

$$\mathrm{Live}_{\mathcal{C}}(a) \iff a \notin W \ \wedge\ \mathrm{req}(a) = \rho_{\tau(a)} \ \wedge\ \mathrm{LocalOK}(a) \ \wedge \bigwedge\nolimits_{a' \in \mathrm{supp}(a)} \mathrm{Live}_{\mathcal{C}}(a'), \qquad (1)$$

where W is the set of withdrawn acceptances, $\mathrm{req}(a)$ and $\tau(a)$ are the requirements and the task associated with a; an empty delivery is never valid. Because $\mathrm{supp}(a)$ names only acceptances that already exist, the support graph is acyclic and the recursion is well founded (Appendix A.1). $\mathrm{LocalOK}(a)$ holds when the verifier was authorized, the evidence satisfies the rule named by the requirements, such as a passing replay under a specified test, and a depends on every acceptance required by those requirements, such as an accepted reconstruction for packaging. A task's progress is the set of its valid acceptances.

Accept admits a verifier's judgment only when it is bound to the submitted artifact and would create a valid acceptance. The submission names the task, its requirements, the frozen artifact, the evidence,

and the acceptances it depends on, and a verifier designated by the requirements judges the artifact. The kernel checks the judgment against the state in which it takes effect, so a judgment made before the requirements or a supporting acceptance changed is rejected. If the check passes, the kernel adds the acceptance, makes it the current delivery, and marks the task fulfilled in one atomic step, even when it replaces an earlier delivery; acceptances depending on the earlier one remain valid until it is withdrawn. Otherwise, nothing changes except the resources already spent. The kernel thereby rejects every substitute for a current judgment: a diagnostic check run by the harness is not a judgment, a judgment under stale requirements is invalid, and a judgment of one artifact never transfers to another.

Withdraw reopens exactly the tasks whose delivered acceptance has lost its basis. When a verifier or the user withdraws one or more acceptances, the kernel adds them to W, together with every acceptance that depends on them, directly or through others. It then reopens exactly the fulfilled tasks whose delivered acceptance is among them, clearing the delivery and returning the task to outstanding. A reopened task does not fall back to another valid acceptance, because that would change the delivered artifact without a new judgment. This differs from a truth-maintenance system, which preserves a conclusion as long as some justification survives (Doyle, 1979; de Kleer, 1986). When the user changes a task's requirements, all of its acceptances and those depending on them become invalid, and the task returns to outstanding; a release does the same but sets the status to `released`.

Together, these rules establish a single invariant.

Invariant 1 (Harness-independent integrity)**.** *If every write to a Contract passes through K atomically, with authenticated actors and immutable identities (Assumption 1), then from any valid initial state and under any sequence of harnesses, every reachable Contract satisfies $z_\tau = \texttt{fulfilled} \Rightarrow \mathrm{Live}_{\mathcal{C}}(\sigma_\tau)$.*

Both *Accept* and *Withdraw* preserve this invariant, and no other operation writes the acceptances, the delivery, or the status (Appendix A.4). Because only the user can release or revise a task and no operation resets its resource use, a stopped, replaced, or exhausted harness leaves its outstanding responsibility visible rather than silently completing the task. Our prototype uses a coarser task-level withdrawal rule, which preserves the invariant but may reopen additional tasks, exactly the trade-off predicted by Theorem 1 (Appendix B.5). The invariant governs who may record progress, not whether a verifier's judgment is correct.

3.2 Disentangling the Harness for Self-Improvement

Because the Contract holds the responsibility and its progress, nothing that defines the task or its progress lives in the harness. Every harness therefore operates under the same rules, and replacing one mid-task, or compacting, restarting, or delegating cannot change the responsibility or its acceptances. What such changes can break is the harness's own execution state. Thus recovery restores execution for consistency, retains observations for information, and never rolls back the Contract.

At each step, the harness builds a context from its execution state, workspace, and a read-only view of the Contract; the model chooses an action; and the harness executes it, updates its state, and may issue a request to the kernel. That request is the harness's only channel to the Contract, and the kernel applies the same admission rules regardless of which harness issued it (Proposition 4). The invariant therefore holds for any harness, including one written by the agent itself.

Replacing the harness mid-task. A new version of the harness is qualified before use: starting from fresh state, it must build successfully and pass fixed checks. A successor reads the same Contract, so a valid acceptance survives the replacement. Saved observations, such as reference outputs, may also carry over, but conclusions drawn from them about an earlier artifact do not transfer to a revised one. Memory is different: a new harness may store it differently, so it must be migrated, and that migration may fail.

A controller outside the agent therefore activates a new version as a trial from a checkpoint of the old harness, its memory, and its workspace, with the memory migrated, and rolls it back if model or tool activity ceases. This health test checks only activity; quality is evaluated separately when adoption is decided. On rollback, the old harness resumes from the complete checkpoint because its memory was formed over that workspace; the trial's workspace remains available, read-only, since what its

observation may still be useful; and the Contract is never restored. Acceptances recorded during the trial therefore stand, and the restored harness sees them through the Contract rather than through its memory (Corollary 1). Appendix B.4 gives the procedure, and Section 4.3 exercises both paths.

Self-improvement as a responsibility. Because the harness is replaceable, improving it becomes another responsibility the kernel can hold (Figure 1c). The deliverable is the source of a new version, submitted as a frozen artifact and judged like any other. The version whose code performs this work, including its instructions for producing successors, is the *research parent*; it works on its own source, within the permitted edits and the available experience, using any model or procedure. What matters is which version's code runs: recursion begins when a generated version becomes the research parent and produces the next version with its own code. The harness under improvement never judges itself: a verifier judges the candidate, the kernel records the judgment, and the user decides adoption. The agent can rewrite how it works but cannot accept, select, or adopt its own revision.

A candidate passes four distinct decisions, none controlled by the candidate: (a) the improvement task's verifier accepts the exact source; (b) the controller builds and qualifies the executable; (c) a rule fixed in advance may select it as the next research parent; and (d) the user may adopt it as the *incumbent*, the version that serves later tasks (Appendix C.2.1). Because (c) and (d) are separate, a candidate can seed further exploration without replacing the incumbent. Parentage itself is not evidence: descent creates no dependency between acceptances, so withdrawing a parent's evaluation does not reopen tasks completed by its descendants. An Evaluation Contract, judged by a separate verifier, records the comparison between the frozen candidate and the incumbent; a Selection Contract, judged by the user, depends on that evaluation and exposes the adopted version to later tasks. A version is adopted exactly while a Selection Contract naming it is fulfilled, and if the evaluation loses its basis, Withdraw reopens the Selection Contract like any other task (Corollary 2).

4 Empirical Evaluation on Long-Horizon Tasks

To evaluate ProContract, we ask three questions:

- **RQ1:** Does the kernel keep agents from diluting responsibility and misjudging progress?
- **RQ2:** Can the agent improve its own harness without losing the task?
- **RQ3:** Does the gain hold for frontier models and on tasks released after the model was trained?

4.1 Experimental Setup

Systems. RQ1 and RQ2 run GPT-5.6 Terra at max reasoning effort on the 200 ProgramBench reconstruction tasks (Yang et al., 2026). *Base* is ProContract before self-improvement: it runs the solving instructions of v_0, the version self-improvement starts from, with earlier harness code. Two rounds of self-improvement, generated with GPT-5.6 Luna and reported in full in Appendix C.2, produce v_1 and then v_2. RQ1 compares base with the public mini-SWE-agent run of the same model (Vals AI, 2026). RQ2 runs v_2 on the same tasks. RQ3 runs GPT-5.6 Luna and Sol on the official ProgramBench leaderboard and runs Terminal-Bench 2.0 with GPT-5.2 against its public leaderboard (Merrill et al., 2026).

Metrics. For each task we record the fraction of hidden tests that the delivered artifact passes. A task is *resolved* when it passes every test; following the public ProgramBench run pages, we also count tasks that pass at least 95%. The mean pass rate averages all 200 tasks, counting an unscored task as zero. A delivery is valid when the kernel has accepted the delivered artifact and the benchmark can score it. On ProgramBench, the verifier accepts an artifact that rebuilds and passes the regression suite the agent wrote; the hidden tests measure behavior beyond that suite. Cost is the model cost per task: model costs use published Standard prices. A point is on the empirical Pareto frontier if no other point costs no more and scores at least as much, with one strict improvement. Paired intervals resample tasks (Efron, 1979).

4.2 RQ1: Raising the Floor

With the same model, ProContract has a much thinner lower tail (Figure 2), and its execution records show the kernel blocking both failure modes where they arise. With the same model, base

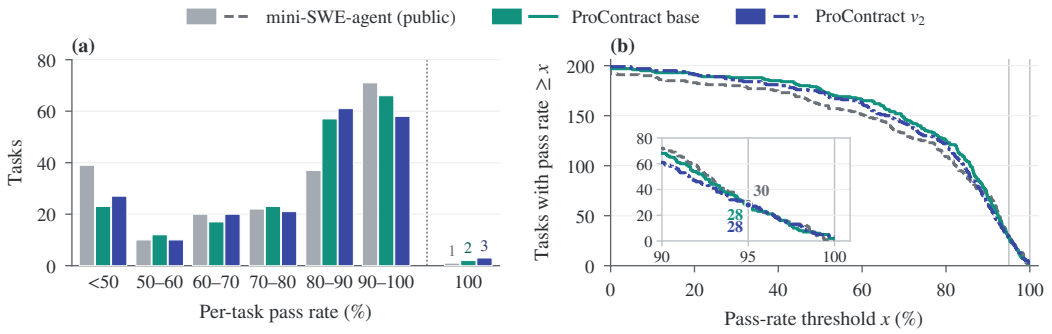

Figure 2: Per-task hidden-test pass rates on the 200 ProgramBench tasks with GPT-5.6 Terra (max): mini-SWE-agent and ProContract base (RQ1), and v_2 (RQ2). (a) Tasks per pass-rate bin; the separate bin on the right counts resolved tasks. (b) Tasks with pass rate at least x; vertical lines mark 95% and 100%, and the inset enlarges $x \geq 90$. Unscored tasks count as zero (three for base, one for v_2).

averages 77.24% against 72.35% for mini-SWE-agent. On the 197 tasks both score, it is higher on 111 and lower on 78, a paired gain of 5.33 points (95% interval 2.90 to 7.95). The gain sits at the bottom: 23 tasks score below 50% against 39, and 5 below 10% against 10. At the top it vanishes: 28 tasks pass at least 95% against 30, and base resolves 2 against 1. The model sets the ceiling; the kernel raises the floor.

The tasks at mini-SWE-agent's floor are tasks the model can solve. Eight of its 10 collapses score exactly zero. `masscan` scores zero for most models and `pandoc` is hard for every system, but on five of the other six, base scores between 76% and 93%, and on `proj` it reaches 38%.

The kernel blocks both failure modes where they would arise. Progress misjudgment arises when a passing check is remembered as done; in the kernel the check is only evidence. In the v_2 run, all 420 passing checks left their tasks open, and 145 of the 199 tasks with one went on to deliver a later artifact. Scored against the first artifact whose own check passed, the artifact finally delivered is better on 65 of 140 scored tasks and worse on 14 (mean +1.1 points, 95% interval 0.3 to 2.3); on `hush` it rises from 23% to 88%. It also arises when a final message is remembered as a delivery; in the kernel delivery is the accepted artifact, and in all 198 deliveries the submission, the verifier's report, and the exported archive name the same artifact. Responsibility dilution arises when a server restart erases the context and the requirements with it; in the kernel they live in the Contract, and all three tasks that resumed in a restarted harness kept their requirements and deadline and delivered.

The floor has a price. Base costs $13.50 per task against $5.38 for mini-SWE-agent: it keeps working, and it still carries its working history in context, so a fifth of its requests exceed 272K input tokens, OpenAI's long-context price tier. The responsibility and its progress no longer need that context. RQ2 asks what the agent does with that freedom.

4.3 RQ2: Improving the Harness Without Losing the Task

Since the harness holds neither the responsibility nor its progress, the agent can rewrite it. Each round runs the current research parent on an improvement task (Section 3.2): v_0 generates v_1, and v_1 generates v_2 with its own code. Both rounds changed both parts:

Round	Harness instructions	Harness code
$v_0 \rightarrow v_1$	evidence-led search with recall of stored observations	enabled observation packing, extended to structured tool results
$v_1 \rightarrow v_2$	rewritten around repair, contrast, and recall	compaction keeps diagnostic records

Both rounds edited the same thing: what the harness keeps in context. The first harness edit moved tool output out of the context window into records the agent can recall, the second made compaction keep the diagnostic records, and each instruction edit taught the agent to work from those records. The agent applied to its own evidence the rule the kernel applies to responsibility and progress: keep

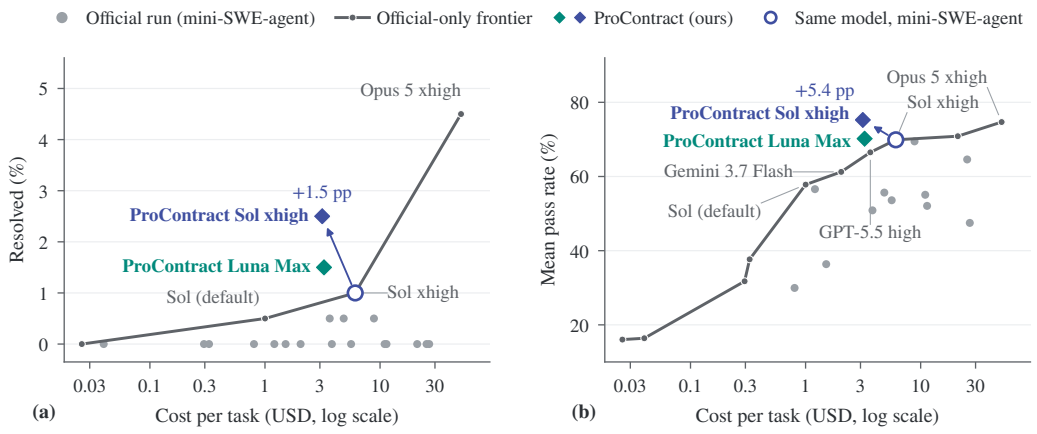

Figure 3: Official ProgramBench leaderboard: (a) resolved rate and (b) mean pass rate against reported cost per task. Gray points are the 21 official entries, all run with mini-SWE-agent, and the line is their frontier. Diamonds mark the Luna and Sol ProContract configurations; the arrow links the official GPT-5.6 Sol xhigh entry to ProContract with the same model. Costs are as reported by each source; Terra results appear in Figure 2.

them out of the context. The edit is safe only under the kernel. Without it, the context holds the responsibility and its progress, and whatever compaction drops is lost from the task.

Exploring from a version is not adopting it, and the kernel keeps that decision outside the version. v_2 raised the mean pass rate by 6.37 points over v_0 on the six development tasks that choose the next research parent, but lost 0.23 points on the three confirmation tasks that decide adoption. The incumbent therefore stayed v_0, and v_2 became only the next research parent.

Unadopted, v_2 can still run for evaluation, because a generated harness can only request, never write. It ran unchanged on all 200 Terra tasks with every Contract intact: 32 of 33 recalls of packed observations returned their stored bytes, and its one completed compaction kept the diagnostic records it was written to keep. In scripted-provider tests, installing a rebuilt harness mid-task and rolling back a stalled one leave the Contract untouched (Appendix C.2.6).

Against base, v_2 keeps the floor at about half the cost (Figure 2). Cost per task falls from \$13.50 to \$7.22 and requests above 272K input tokens from 8,481 to 898, while tasks below 10% go from 5 to 4, valid deliveries from 195 to 198, and resolved tasks from 2 to 3. The price is paid in the upper middle of the distribution. On the tasks both score, the mean changes by -2.16 points (95% interval -3.31 to -1.03), and the loss sits where contexts shrank: -5.3 points in the quarter whose median context shrank most, against $+0.5$ where it did not shrink. Context caches capability, not the task.

4.4 RQ3: Generalization and Post-Cutoff Tasks

Other frontier models gain from the kernel too, and the gain holds on tasks released after the model was trained. On the official ProgramBench leaderboard, where every entry runs mini-SWE-agent, GPT-5.6 Sol at xhigh averages 69.89%; ProContract with the same model and effort averages 75.25%. It is higher on 108 tasks and lower on 90, a paired gain of 5.36 points (95% interval 2.46 to 8.39), and again the gain is at the floor: 3 tasks score below 10% against 15, and 29 below 50% against 42, while 30 pass at least 95% against 31. This is the signature of RQ1 with a second model. Across the leaderboard (Figure 3), ProContract Sol has the highest mean pass rate of all 23 points and resolves five tasks against two for the same model; the only entry that resolves more, Claude Opus 5 at xhigh, costs 16 times as much per task.

Post-cutoff tasks. To test on tasks newer than the model, we pair Terminal-Bench 2.0 with GPT-5.2: the benchmark was released after the model's published knowledge cutoff, and GPT-5.2 has the most public harnesses on its leaderboard. ProContract passes 65.7% $\pm$ 1.1 of the 89 tasks over two trials, the highest of the five, against 64.9% for Droid (Factory, 2025), 62.9% for Codex CLI (OpenAI, 2025),

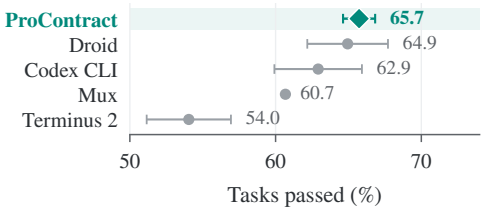

Figure 4: Terminal-Bench 2.0 with GPT-5.2 on all 89 tasks. Gray points are the public leaderboard entries with that model and their reported 95% intervals where available; ProContract averages two trials, with the same interval.

60.7% for Mux (Coder, 2025), and 54.0% for Terminus 2 (Merrill et al., 2026), the benchmark's own harness (Figure 4). Its harness code carries the diagnostic compaction that self-improvement added in v_2.

5 Related Work

Keeping the task across a long horizon. Long-horizon agents keep their task in the context and memory their harness manages: MemGPT pages information between context and external storage (Packer et al., 2023), Reflexion carries verbal feedback across trials (Shinn et al., 2023), and coding harnesses such as SWE-agent and OpenHands keep an event history the agent reads back (Yang et al., 2024; Wang et al., 2025). Durable workflows and checkpoint recovery persist execution state across failures (Temporal Technologies, 2026; Wu et al., 2026c). Each makes the agent's working material last longer but leaves the agent to decide what is owed and when it is done; ProContract keeps the responsibility outside every harness and admits progress only as a verifier's judgment. LongHorizon-Harness comes closest: it keeps task state outside the execution context, where a model-driven manager maintains it and admits only facts that a read-only auditor has verified (Ma et al., 2026). In ProContract a deterministic kernel rather than a model holds that state, an acceptance counts only while its basis holds, and the harness itself can be replaced or improved mid-task. ECT and Matrix add evidence-based completion and support-dependent recovery (Liu, 2026; Salas, 2026), and truth-maintenance systems retract conclusions whose support is withdrawn (Doyle, 1979; de Kleer, 1986); in ProContract a task that loses its support reopens rather than switching silently to another acceptance.

Improving the harness. Several systems revise the code that runs an agent. DGM keeps an archive of self-modified agents as parents for further evolution (Zhang et al., 2026); Meta-Harness builds new harnesses from earlier code, scores, and traces (Lee et al., 2026); ModularRSI revises individual modules from contrasts between successful and failed runs (Wu et al., 2026b); and SoL-Pi evaluates frozen candidates on isolated held-out tasks (Liu et al., 2026). Aspire and HarnessDev study how such evolution generalizes across goals and models (Wu et al., 2026e;d). These methods decide which version to keep; ProContract makes each improvement one more responsibility under the same kernel, so a new version can replace the running harness mid-task, and exploring from a version stays separate from adopting it. Their evaluators can serve as ProContract's verifiers.

6 Conclusion

Long-horizon agents lose tasks their model can solve because the responsibility and its progress live in the execution state they rewrite. ProContract decouples both from the harness: only the kernel can change the Contract that holds the responsibility, and progress enters only as an authorized judgment of an exact artifact. Improving the harness then becomes one more task whose adoption stays outside the agent. With the same model, the kernel raises the floor, not the ceiling; the agent's own revision moved evidence out of its context and ran all 200 tasks under the kernel; and the gains carry to GPT-5.6 Sol and Terminal-Bench 2.0. An agent can thus change how it works as often as it needs to, without changing what it owes or who decides that it is done.

Limitations. ProContract guarantees integrity, not correctness: it decides who may record progress, not whether the specification is complete or the verifier right, so a strong harness can no longer mask a weak verifier. Our self-improvement experiment shows controlled self-modification rather than monotonic improvement: a generated version rewrote its own harness without being able to adopt itself. The prototype withdraws at Contract rather than record granularity; it still preserves the invariant and errs only toward reopening more, the conservative corner of the design space that Theorem 1 characterizes. Appendix E discusses the remaining assumptions.

AI USE STATEMENT

AI assistants supported manuscript editing and data-analysis code. Authors remain responsible for the measurements, reasoning, and final claims.

ETHICS STATEMENT

This work studies how long-horizon agents keep their assigned responsibilities and record their progress. It involves no human subjects and no personal data: all experiments run on public benchmarks, and every comparison point comes from a public leaderboard. The agents execute code autonomously and, during self-improvement, write code that later runs as their own harness. PROCONTRACT keeps acceptance, selection, and adoption outside the agent, so a generated harness can neither record its own progress nor adopt itself. It does not make a verifier correct or a generated harness safe to execute, so deploying self-improving agents still requires isolated execution and an adoption decision outside the agent.

REPRODUCIBILITY STATEMENT

Section 3 and Appendix A define the kernel's reference semantics. Appendix A.4 proves Invariant 1, and Appendix A.3 proves Theorem 1 and Proposition 3. Appendix B describes the implementation, including version activation and rollback and the prototype's trust assumptions. Section 4.1 and Appendix C.1 give the models, reasoning efforts, turn limits, parallelism, runtime environment, scoring rules, and cost accounting, and Table 5 lists every official leaderboard entry we compare against. Appendix C.2 gives the self-improvement controller, its development and confirmation tasks, and the results of every generated candidate. We will release the kernel, the harness versions, and the per-task results upon publication.

REFERENCES

Laksh Advani. From Confident Closing to Silent Failure: Characterizing False Success in LLM Agents. *arXiv preprint arXiv:2606.09863*, 2026. URL https://arxiv.org/abs/2606.09863v1.

Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *The Journal of Symbolic Logic*, 50(2):510–530, 1985. doi: 10.2307/2274239. URL https://www.cambridge.org/core/product/identifier/S0022481200032849/type/journal_article.

Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4): 181–185, 1985. doi: 10.1016/0020-0190(85)90056-0. URL https://www.cs.cornell.edu/fbs/publications/DefLiveness.pdf.

Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, and Munindar P. Singh. Composing and verifying commitment-based multiagent protocols. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*, pp. 10–17, 2015. ISBN 9781577357384. URL https://www.ijcai.org/Proceedings/15/Papers/009.pdf.

Evan Chen, Shiqiang Wang, and Christopher G. Brinton. Fresh Memory, Stale Plans: Dependency-Scoped Validation for Distributed LLM-Agent Memory. *arXiv preprint arXiv:2609.03340*, 2026a. URL https://arxiv.org/abs/2609.03340v1.

Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, Haiming Zheng, and Qing Wang. From Failed Trajectories to Reliable LLM Agents: Diagnosing and Repairing Harness Flaws. *arXiv preprint arXiv:2606.06324*, 2026b. URL https://arxiv.org/abs/2606.06324v2.

Coder. Mux, 2025. URL https://github.com/coder/mux.

Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986. doi: 10.1016/0004-3702(86)90080-9.

Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979. doi: 10.1016/0004-3702(79)90008-0.

Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. Generalization in adaptive data analysis and holdout reuse. In *Advances in Neural Information Processing Systems*, volume 28, 2015. URL https://papers.nips.cc/paper/2015/hash/bad5f33780c42f2588878a9d07405083-Abstract.html.

B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1):1–26, 1979. doi: 10.1214/aos/1176344552. URL https://doi.org/10.1214/aos/1176344552.

Factory. Droid, 2025. URL https://factory.ai/.

Sanford J. Grossman and Oliver D. Hart. The costs and benefits of ownership: A theory of vertical and lateral integration. *Journal of Political Economy*, 94(4):691–719, 1986. doi: 10.1086/261404.

Oliver Hart and John Moore. Property rights and the nature of the firm. *Journal of Political Economy*, 98(6):1119–1158, 1990. doi: 10.1086/261729.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. doi: 10.1145/78969.78972. URL https://cs.brown.edu/people/mph/HerlihyW90/p463-herlihy.pdf.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2408.08435.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Sajjad Khan. Resume Means Resume: A Machine-Checked Conformance Contract for Checkpoint, Interrupt, and Resume Semantics in Workflow Persistence Layers. *arXiv preprint arXiv:2608.03836*, 2026. URL https://arxiv.org/abs/2608.03836v3.

Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 9780321143068.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026. URL https://arxiv.org/abs/2603.28052.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses. *arXiv preprint arXiv:2604.25850*, 2026. URL https://arxiv.org/abs/2604.25850v4.

Haozhe Liu, Tian Ye, Sensen Gao, Qihang Cao, Yitong Li, Mingchen Zhuge, Duomin Wang, Ruihua Zhang, Jiawang Bian, Lei Zhu, Ligeng Zhu, Enze Xie, and Song Han. SoL-Pi: Scaling auto-research loops for efficient agent harnesses. Project technical report, 2026. URL https://nvlabs.github.io/SoL-Pi/. Project page accessed September 20, 2026.

Jason Liu. When May an Agent Stop? Evidence-Carrying Termination for Tool-Using LLMs. *arXiv preprint arXiv:2608.23623*, 2026. URL https://arxiv.org/abs/2608.23623v1.

Yongjian Lyu, Yang Ren, Ruofei Lai, and Wenting Liu. From Version Conflicts to Decision Conflicts: Selective Revalidation for Long-Running AI Agents. *arXiv preprint arXiv:2609.08015*, 2026. URL https://arxiv.org/abs/2609.08015v1.

Ziyu Ma, Hailang Huang, Shun Zou, Yong Wang, Shidong Yang, Yiming Hu, Fei Wei, and XiangXiang Chu. LongHorizon-Harness: Advancing long-horizon agents for real-world tasks. *arXiv preprint arXiv:2608.01964*, 2026. URL https://arxiv.org/abs/2608.01964.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. Build systems à la carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP):79:1–79:29, 2018. doi: 10.1145/3236774.

Michael Nguyen, Quoc Nguyen, and Paul Vuong. Recursive Self-Evolving Agents via Held-Out Selection. *arXiv preprint arXiv:2606.28374*, 2026. URL https://arxiv.org/abs/2606.28374v1.

OpenAI. Codex CLI, 2025. URL https://developers.openai.com/codex/cli/.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. doi: 10.48550/arXiv.2310.08560. URL https://arxiv.org/abs/2310.08560v2. Version 2, revised February 2024.

ProgramBench. Stranger6667/jsonschema: Task results, 2026. URL https://programbench.com/task/stranger6667__jsonschema.d52e881/. Official task page, accessed September 22, 2026.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025. URL https://arxiv.org/abs/2504.15228.

Josh Rosen and Seth Rosen. From Agent Loops to Deterministic Graphs: Execution Lineage for Reproducible AI-Native Work. *arXiv preprint arXiv:2605.06365*, 2026. URL https://arxiv.org/abs/2605.06365v1.

Jesus Salas. Correct Is Not Governed: Provenance Integrity in Agentic Workflows. *arXiv preprint arXiv:2608.12761*, 2026. URL https://arxiv.org/abs/2608.12761v1.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. doi: 10.1109/PROC.1975.9939. URL https://web.mit.edu/Saltzer/www/publications/protection/.

Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000. doi: 10.1145/353323.353382. URL https://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf.

Jiajun Shi, Siyuan Tao, Yuhao Wu, Zexuan Wang, Jingyuan Zhang, Jiaheng Liu, Xinping Lei, Xinrong Zhang, Siyuan Fang, Zhewen Tan, Tianle Cai, Junhao Fang, Jiameng Huang, Yueyang Wang, Jinkai Liu, Yuxuan Zhang, Jian Yang, Zhoujun Li, Shen Yan, Wenhao Huang, and Ge Zhang. S3Gym: Can LLMs turn self-testing and self-judging into self-improvement? *arXiv preprint arXiv:2608.31100*, 2026. URL https://arxiv.org/abs/2608.31100v1.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Munindar P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7(1):97–113, 1999. doi: 10.1023/A:1008319631231.

Temporal Technologies. Temporal workflow execution overview. Temporal Documentation, 2026. URL https://docs.temporal.io/workflow-execution. Accessed 2026-09-07.

The Kubernetes Authors. Admission control in Kubernetes. Kubernetes Documentation, 2026a. URL https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/. Accessed 2026-09-22.

The Kubernetes Authors. Controllers. Kubernetes Documentation, 2026b. URL https://kubernetes.io/docs/concepts/architecture/controller/. Accessed 2026-09-22.

Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium*, pp. 1393–1410. USENIX Association, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

Vals AI. ProgramBench: Rebuilding programs from binaries. Vals AI benchmark website, 2026. URL https://www.vals.ai/benchmarks/programbench. Updated 2026-09-16; accessed 2026-09-18.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. URL https://arxiv.org/abs/2305.16291v2.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the Evaluation of Harness Evolution for Agents. *arXiv preprint arXiv:2607.12227*, 2026a. URL https://arxiv.org/abs/2607.12227v1.

Zehao Wang, Yisen Xu, Chenglin Li, Chao Peng, Bram Adams, Ahmed E. Hassan, and Tse-Hsun Peter Chen. Turning interaction history into execution state: A runtime layer for long-horizon coding agents. *arXiv preprint arXiv:2608.00808*, 2026b. URL https://arxiv.org/abs/2608.00808.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-Memory: Benchmarking LLM agent test-time learning with self-evolving memory. *arXiv preprint arXiv:2511.20857*, 2026. URL https://arxiv.org/abs/2511.20857v2. Version 2, revised May 18, 2026.

Guanlong Wu, Dahui Li, Ke Jiang, Jianyu Niu, Cong Wang, and Yinqian Zhang. Safe to Resume? Breaking Execution Continuity of Agent Execution via Rollback. *arXiv preprint arXiv:2608.29381*, 2026a. URL https://arxiv.org/abs/2608.29381v1.

Siwei Wu, Jincheng Ren, Yizhi Li, Haau-Sing Li, Chengran Yang, Yuxuan Zhang, Weicheng Gu, Jian Yang, Riza Batista-Navarro, Chuanyi Zhang, Xianglong Liu, Ming Zhou, Bryan Dai, and Chenghua Lin. ModularRSI: Modular and generalizable recursive harness self-improvement. *arXiv preprint arXiv:2609.14857*, 2026b. URL https://arxiv.org/abs/2609.14857.

Tianyuan Wu, Chaokun Chang, Lunxi Cao, Wei Gao, and Wei Wang. Crab: A Semantics-Aware Checkpoint/Restore Runtime for Agent Sandboxes. *arXiv preprint arXiv:2604.28138*, 2026c. URL https://arxiv.org/abs/2604.28138v1.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, Zhiyuan Zeng, Shaowen Wang, Jinkai Liu, Yunfeng Shi, Jiaheng Liu, Shen Yan, Wenhao Huang, Ge Zhang, and Wenxuan Zhang. HarnessDev: Can LLMs Create and Evolve Their Own Agent Harness? *arXiv preprint arXiv:2609.01437*, 2026d. URL https://arxiv.org/abs/2609.01437v1.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Yuxuan Zhang, Xinping Lei, Junting Zhou, Zexuan Wang, Yuchen Wu, Huan Zhou, Duo Wang, Yinzhu Piao, Yongchang Peng, Yunfeng Shi, Jin Chen, Zuo Wang, Jinkai Liu, Jiaheng Liu, Wenxuan Zhang, Shen Yan, Wenhao Huang, and Ge Zhang. Aspire: Can models self-evolve from vague goals? *arXiv preprint arXiv:2608.31111*, 2026e. URL https://arxiv.org/abs/2608.31111v1.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent–computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026. URL https://arxiv.org/abs/2605.03546.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629v3.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2310.02304.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2505.22954.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160. PMLR, 2024. URL https://proceedings.mlr.press/v235/zhou24r.html.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, Yueyang Wang, Junfeng Zhong, Rui Liu, Tian Gao, Yangguang Lin, Jingyuan Zhang, Maojia Song, Xuan Qi, Jinhong Wu, Chenyang Zhang, Yinzhu Piao, Ziru Niu, Hongbin Lin, Lingxiang Meng, Peng Tang, Chengyao Tang, Shanyu Wu, Huanyu Zheng, Yu Liu, Liya Zhu, He Wang, Ming Ding, Ziyu Wan, Hao Liu, Sibo Wang, Haotian Zhu, Xintian Zhang, Nan Chai, Yipeng Liu, Panhao Lai, Sihang Yuan, Zixin Su, Ge Zhang, Wangchunshu Zhou, Yantao Du, Wenhao Huang, and Guang Shi. EdgeBench: Unveiling Scaling Laws of Learning from Real-World Environments. *arXiv preprint arXiv:2607.05155*, 2026. URL https://arxiv.org/abs/2607.05155v1.

APPENDIX CONTENTS

A Formal Analysis and Discussion **15**

A.1 The reference model 16

A.2 Atomic transitions and dependent invalidation 17

A.3 What progress must record 19

A.4 Integrity under any harness 22

A.5 Replacing harnesses 23

A.6 Self-improvement through the kernel 24

A.7 Discussion 25

B Implementation Details **26**

B.1 State ownership and authority 26

B.2 Scheduling and the implemented continuation paths 26

B.3 Acceptance and withdrawal control 27

B.4 Version activation and rollback 28

B.5 Implementation scope and trust assumptions 29

C Extra Experimental Results **29**

C.1 Evaluation protocol and supplementary results 29

C.2 Recursive agent revision 34

C.3 Terminal-Bench 2.0 37

C.4 The jsonschema example 38

D Extra Related Works **39**

D.1 Execution state, completion, and recovery 39

D.2 Harness improvement and candidate selection 40

E Limitations **42**

APPENDIX GUIDE

Appendix A gives the full state, transitions, assumptions, and proofs behind the kernel's properties. Appendix B describes the benchmark harnesses' task interfaces and recovery paths, and their differences from the reference model. Appendix C.1 gives the configurations and accounting behind RQ1 to RQ3, including the public comparison table. Appendix C.2 records the RQ2 candidates, lineage, and Luna-study adoption rules. Appendix C.3 gives the Terminal-Bench 2.0 protocol, and Appendix C.4 follows one Terra task in detail. Appendix D compares the execution and improvement mechanisms of related systems, and Appendix E discusses limitations.

A FORMAL ANALYSIS AND DISCUSSION

This appendix follows the order of Section 3. Appendices A.1 and A.2 define the reference model of the kernel, Appendix A.3 shows why progress must be recorded as judgments about exact results, and Appendix A.4 proves Invariant 1. Appendices A.5 and A.6 derive what the kernel implies for

replacing harnesses and for recursive self-improvement, and Appendix A.7 discusses what these results do and do not establish. The model is a specification; Appendix B.5 compares it with the prototype.

Notation. We keep the notation of Section 3.1. The kernel's state $\mathcal{C}$ is the set of Contracts, one for each issued task τ in a finite set $\mathcal{T}_{\mathcal{C}}$, and the Contract of τ stores the current requirements ρ_τ, the budget b_τ and cumulative resource use $u_\tau \leq b_\tau$, acceptance records A_τ, withdrawn records $W_\tau \subseteq A_\tau$, a selected record $\sigma_\tau \in A_\tau \cup \{\bot\}$, and a status z_τ. The reference model refines the main text's outstanding status into `open` and `submitted`, the latter while a submission awaits judgment. The reference model adds three operational fields: a pending submission q_τ, a responsible route π_τ naming who must act on outstanding work, either a harness queue or the user's resolution queue, and a release certificate for a released task. A revision ρ is an immutable object that fixes the acceptance conditions, including required prerequisites, and the rule for admitting evidence; changing any of them creates a fresh revision identifier. The *user* issues, revises, and releases tasks. A *verifier* is a role authorized to judge submissions, and a *judgment* is its authenticated statement.

Definition 1 (Acceptance and release)**.** *An acceptance of (τ, ρ, x) is an authorized decision that the admitted evidence for the exact artifact x meets the acceptance conditions of the current revision $\rho = \rho_\tau$. User-authorized release ends the duty without asserting success.*

In the reference model, release also makes the task's acceptance records ineligible for future reliance. This coupling is a protocol choice, not a consequence of what release means. A reported counterexample does not by itself withdraw an acceptance; an authorized decision must find that the support no longer suffices. The results below begin at that decision and concern its consequences, not how counterexamples are found.

Assumption 1 (Kernel-mediated updates)**.** *Authority-bearing events are authenticated, and every authoritative write passes through the kernel K. Revocation of authority or of an admission rule enters only through an admitted revision or withdrawal. Identifiers resolve to immutable objects. Guards read the state in which a decision takes effect, and the decision, its state changes, and any mandatory dependent invalidation are atomic.*

A.1 The reference model

At a transition boundary, $\mathcal{C}$ contains the tasks with the fields above, immutable objects such as revisions, artifacts, and evidence, the acceptance records, and an append-only decision history. Resource quantities may be vectors compared componentwise. The kernel never resets u_τ or replaces b_τ on a harness change or a revision; additional budget grants are outside this transition vocabulary. Escalation changes the route π_τ, not the status.

Submissions and records. A pending submission $q_\tau = (\tau, \rho, x, e, D)$ names the current revision, an exact artifact x, evidence e, and the identifiers D of the records claimed as support. A submission does not change the recognized subject: a fulfilled task may have a pending replacement submission while its selected record remains in force. For an outstanding task, submission changes open to submitted.

An acceptance record a has a fresh identifier, an owning task $\tau(a)$, the requirements $\mathrm{req}(a)$ it was judged under, an artifact $x(a)$, evidence $e(a)$, a judgment $j(a)$, and a finite support set $\mathrm{supp}(a)$. Its coordinate is

$$\kappa(a) = \big(\tau(a), \mathrm{req}(a), x(a), e(a)\big).$$

The coordinate names immutable objects; a mutable workspace cannot stand in for $x(a)$. The judgment binds the identifier, the whole coordinate, and the exact set $\mathrm{supp}(a)$, since binding only $\kappa(a)$ would leave the support edges open to substitution. A record's body and edges never change. Withdrawal adds the record to $W_{\tau(a)}$ and appends a decision; it does not delete the judgment, and a withdrawn record never becomes eligible again. Considering the same artifact or evidence again requires a fresh record and a new decision.

Support and liveness. Let $A = \bigcup_\tau A_\tau$ and $W = \bigcup_\tau W_\tau$. The support graph $G_{\mathcal{C}} = (A, E)$ has an edge $(a', a) \in E$ exactly when $a' \in \mathrm{supp}(a)$, so edges run from a supporting record to the record that relies on it. Dependencies are conjunctive: each listed record is necessary. $\mathrm{Required}_\rho(D)$ checks

that D supplies every prerequisite fixed by ρ, including the required tasks, their revisions, and any restriction on their subjects, so a harness cannot omit a difficult prerequisite. $\mathrm{Authorized}(j(a), \tau(a))$ holds when $j(a)$ is a valid judgment by a role authorized under $\mathrm{req}(a)$ and binds $(a, \kappa(a), \mathrm{supp}(a))$. $\mathrm{Admit}_{\mathrm{req}(a)}(\kappa(a), \mathrm{supp}(a))$ holds when the frozen admission rule accepts the evidence for those exact objects. It is a rule result, not a finding that the evidence is semantically adequate; a rule may demand an authenticated checker report or explicitly trust a verifier's judgment, and these choices support different claims. The local conditions of Equation 1 are

$$\begin{aligned}\mathrm{LocalOK}(a) \iff\ & \mathrm{Authorized}(j(a), \tau(a)) \wedge \mathrm{Required}_{\mathrm{req}(a)}(\mathrm{supp}(a)) \\ & \wedge \mathrm{Admit}_{\mathrm{req}(a)}(\kappa(a), \mathrm{supp}(a)),\end{aligned}$$

and liveness is evaluated by recursion on $G_{\mathcal{C}}$:

$$\mathrm{Live}_{\mathcal{C}}(a) \iff a \in A \setminus W \ \wedge\ \mathrm{req}(a) = \rho_{\tau(a)} \ \wedge\ \mathrm{LocalOK}(a) \ \wedge\ \textstyle\bigwedge_{a' \in \mathrm{supp}(a)} \mathrm{Live}_{\mathcal{C}}(a'). \quad (2)$$

The empty conjunction is true, and a topological order of the acyclic graph makes the evaluation unique. Liveness here is a predicate on the current state, not temporal liveness in the safety/liveness distinction (Alpern & Schneider, 1985). Neither the status nor the selection enters Equation 2. An older record may therefore remain live after a newer one is selected for the same task, and records that rely on it still name the older subject.

Authority grants and admission rules are frozen within a revision. Any change that removes their force must enter through a revision or an accepted withdrawal, with the dependent updates below. There is no unmediated expiry, external permission change, or mutable oracle that could silently change Equation 2; modeling one would require a transition with the same dependent updates.

Stored status and validity. The status z_τ is stored, not computed from liveness. A valid state satisfies the property of Invariant 1 with ownership made explicit,

$$z_\tau = \texttt{fulfilled} \implies \sigma_\tau \in A_\tau \ \wedge\ \mathrm{Live}_{\mathcal{C}}(\sigma_\tau), \quad (3)$$

together with the following conditions. $\sigma_\tau = \bot$ unless τ is fulfilled. Outstanding tasks, those open or submitted, each have a responsible route. A submitted task has a pending submission bound to its current revision, and an open or released task has none. A released task has a user-authorized release certificate and no eligible record of its own, so release cannot serve as prerequisite success. Coordinates are well formed, support identifiers exist, $G_{\mathcal{C}}$ is acyclic, and $0 \le u_\tau \le b_\tau$. The history keeps accepted decisions, earlier revisions, resource charges, and withdrawn records. The empty state is valid; a nonempty initial state must satisfy these conditions, and an arbitrary imported database is not assumed valid.

A.2 Atomic transitions and dependent invalidation

A request names an operation, its actor, and a payload. Table 1 defines the allowed transitions; no other authoritative mutation exists. Every guard reads the state in which the decision takes effect, not an earlier harness view. A failed guard changes no task, record, or resource field, although a rejection may report the failed guard as feedback, and work already done stays charged. Harness authority covers delegated work and submissions. Acceptance, withdrawal, revision, and release require the separately authorized verifier or user. The table defines a relation because routing leaves choices; the kernel K of Section 3.1 is a deterministic implementation of it.

Invalidation. For a set $B \subseteq A$ of records, its inclusive dependent closure is

$$\mathrm{Desc}^*(B) = B \ \cup\ \{a \in A : \text{ some } a' \in B \text{ reaches } a \text{ in } G_{\mathcal{C}}\}. \quad (4)$$

The atomic operation $\mathrm{Inv}(B)$ does the following:

1. It adds every record of $\mathrm{Desc}^*(B)$ to W, traversing the whole historical graph without stopping at records already withdrawn or at tasks whose harness is stopped or escalated.
2. For every task whose selected record lies in $\mathrm{Desc}^*(B)$, it clears σ_τ and q_τ, sets z_τ to open, and keeps or assigns a responsible route.
3. It cancels every pending submission whose support set meets $\mathrm{Desc}^*(B)$. A submitted task returns to open, a task with an unaffected selection stays fulfilled, and the lost support identifier is returned as feedback or, if protected, routed to the user.

Every task that was outstanding stays outstanding and routed. A task whose selected record lies outside the closure stays fulfilled even if an older record of its own is withdrawn. This is not a disjunctive dependency, because the selected record must still satisfy all of its own edges. Record bodies, edges, identifiers, and resource use do not change. Reopening restores the existing responsibility without granting resources: an exhausted task stays outstanding and is routed for resolution, never silently fulfilled or released.

Remark 1 (Order independence)**.** *For a fixed graph, $B \mapsto \mathrm{Desc}^*(B)$ is a closure operator: it is extensive, monotone, and idempotent, and $\mathrm{Desc}^*(B_1 \cup B_2) = \mathrm{Desc}^*(B_1) \cup \mathrm{Desc}^*(B_2)$. Applying Inv to B_1 and then to B_2, with no other change in between, therefore yields the same withdrawal marks, selections, and statuses as the reverse order or a single application to $B_1 \cup B_2$, and the set of reopened tasks grows monotonically with the seeds. The property concerns propagation over fixed seeds, not equality of complete histories: an earlier withdrawal can make a later request fail its liveness guard.*

Table 1: Reference transitions. All guards read the current atomic state. Fields not mentioned remain unchanged. Routing never creates fulfillment or release.

Transition	Guard	Authoritative effect
Issue	Authorized user; fresh task identifier; well-formed revision, route, and budget.	Add τ with $z_\tau = \texttt{open}$, no selection or submission, $u_\tau = 0$, and its ceiling b_τ. Other tasks are unchanged.
Work or check	Delegated operation on a task that is not released; a reservation $d \geq 0$ with $u_\tau + d \leq b_\tau$.	Charge d. Work may edit the workspace; a check may append a receipt for an exact subject. Neither changes selection, support, or status.
Submit	Authorized execution route; task not released; exact immutable artifact and evidence; current revision; existing support records.	Store q_τ. An outstanding task becomes submitted; a fulfilled task keeps its selected record until a new decision.
Accept	Pending submission; fresh record; a judgment bound to that submission whose resulting record would be live (Section 3.1).	Append a and the edges $a' \to a$ for $a' \in \mathrm{supp}(a)$; select a; set fulfilled; clear q_τ. Older records and their edges do not rebind.
Stop, replace, escalate, or exhaust	Authorized scheduling event; outstanding work keeps a responsible route.	Change the harness or route only. Status, selection, revision, identity, and resource use are preserved; exhaustion is not release.
Withdraw	Authorized withdrawal naming an existing eligible record and its exact coordinate.	Apply $\mathrm{Inv}(\{a\})$, with all dependent updates, in this transaction. No harness report alone authorizes withdrawal.
Revise	User-authorized revision of a task that is not released; fresh revision identifier.	Apply $\mathrm{Inv}(A_\tau)$, install the revision, clear selection and submission, set open, and keep a route. Budget and use are preserved.
Release	User-authorized release of a task that is not released, naming its current revision.	Apply $\mathrm{Inv}(A_\tau)$, clear selection and submission, and set released with a certificate. Dependent tasks receive no certificate.
Reject or give feedback	An authorization, coordinate, evidence, resource, or support guard fails.	Preserve every authoritative field. Report the failed guard, or route sensitive evidence to the user.
Read or deliver	One authoritative state; reuse requires a live record; delivery requires a fulfilled task.	Return the selected immutable artifact or the scoped query. Never substitute the workspace, and never treat an old read as permission at a later state.

Acceptance and replacement. The accept transition builds the record from the pending submission rather than from a harness-supplied status. Its support records already exist, so the fresh node has only

incoming edges and the graph stays acyclic. Accepting a new artifact does not retire older records. If a' supports a and a newer a'' is later selected for $\tau(a')$, the edge $a' \to a$ remains, with $x(a')$ as its subject; retiring a' takes an authorized withdrawal of its closure. A system that always retires prior support on replacement can compose withdrawal and acceptance atomically, but it must check the new support after the withdrawal, not in the state before it.

Revision and release. Both operations seed invalidation with *every* record of the task, not only the selected one; otherwise an older acceptance could keep supporting dependents after the requirements changed or the duty ended. A revision installs a fresh identifier even when its text or the artifact bytes are unchanged. Release creates no record and cannot satisfy $\mathrm{Required}_\rho$ in place of a live one. Historical edges never change: a revision may permit different support for future acceptances but does not redirect old records. A dependent task whose revision still requires the old one stays unsatisfied until the user revises or releases it, and releasing an unaccepted prerequisite leaves its dependents outstanding. These rules can leave work blocked; they do not promise a feasible resolution.

Quiet scopes. For a scope $\mathcal{O} \subseteq \mathcal{T}_\mathcal{C}$,

$$\mathrm{Quiet}_\mathcal{C}(\mathcal{O}) \iff \forall \tau \in \mathcal{O} : \; z_\tau \in \{\texttt{fulfilled}, \texttt{released}\}. \tag{5}$$

The scope and the state are part of the answer. An empty scope is vacuously quiet, and nothing follows about tasks outside the scope or issued later. A stopped, submitted, escalated, or exhausted task is outstanding and therefore not quiet. The query is bookkeeping, not a liveness theorem.

A.3 What progress must record

Section 3.1 argued that progress cannot be kept as a memory of what happened. The three results below make this precise: observations do not determine what is owed, the kernel's record-level rule reopens exactly the tasks that lost support, and no rule that sees only task-level state can do so.

A.3.1 Observations do not determine what is owed

Proposition 1 (Observations do not determine what is owed)**.** *Let $V(h)$ be the representation available to an updater after execution prefix h, and suppose $V(h) = V(h')$ while the same admitted request requires different statuses after h and after h'. Then no rule that decides from V and the request is correct on both prefixes. Such pairs exist whenever V retains only artifacts, check results, and harness context.*

Proof of Proposition 1. Equal inputs give the same output, or for a randomized rule the same output distribution, which cannot match two different required statuses with certainty. For existence, let h end with a task fulfilled on a live selected record, and let h' extend h by an authorized revision of the task's requirements that no harness observes, with no edit and no new check. Artifacts, check results, and harness context are the same after h and h'. A delivery request must succeed after h, however, and must be refused after h', where the task is outstanding again. □

This is the formal content of the claim in Section 3.1 that two histories can leave the same visible context while differing in what remains owed. It does not require a separate field for every fact. A compact state may summarize them, provided it keeps the distinctions in Table 2.

Resources and responsibility behave the same way. With a ceiling of ten units, a harness replaced after nine units of use cannot take a two-unit action, whereas one replaced after zero units can, although the artifacts and checks are identical. An idle harness after release and an idle harness after exhaustion also leave different duties. Task state must therefore survive execution changes, in whatever encoding.

A.3.2 Selective reopening

A running example. Return to the command-line tool of Section 3.1 and give it two tasks: τ_1 reconstructs the tool, and τ_2 packages it, where the packaging requirements include an accepted reconstruction. The first reconstruction x_0 is accepted as a_0. The harness then submits a faster reconstruction x_1 with the same behavior, which is accepted as a_1 and selected; a_0 is not withdrawn,

Table 2: Information absent from an artifact and its displayed pass summary. Each distinction can change acceptance eligibility while leaving that visible projection unchanged.

Retained distinction	Why the same visible result can require a different decision
Task and revision	A judgment for a sibling task, an old revision, or a substituted claim cannot fulfill the current task, even if the artifact bytes are identical.
Subject, admission rule, and evidence	A passing display may come from another artifact, a stdout-only observer instead of the required terminal observer, or an unauthenticated report. It identifies neither the checked object nor the admission rule.
Authority and withdrawal	A harness's completion statement lacks the verifier's authority. An authorized acceptance can later be withdrawn without changing its original test observations.
Record identity and support	A dependent may cite a withdrawn record even when a newer live record exists for the same task and artifact. The newer record does not replace the cited support.

because x_0 still met the requirements when judged. The package, a metadata file and an install script that do not change with the reconstruction, is accepted twice with the same artifact and evidence: as b_0 citing a_0 and as b_1 citing a_1. Depending on which of the two acceptances was recorded last, τ_2 selects b_0 or b_1. This gives two states with the same records,

$$\mathrm{supp}(a_0) = \mathrm{supp}(a_1) = \varnothing, \qquad \mathrm{supp}(b_0) = \{a_0\}, \qquad \mathrm{supp}(b_1) = \{a_1\}, \qquad \sigma_{\tau_1} = a_1,$$

which differ only in σ_{τ_2}: it is b_0 in $\mathcal{C}^{(0)}$ and b_1 in $\mathcal{C}^{(1)}$ (Figure 5). The verifier now finds that x_0 was checked against a stale reference output and withdraws a_0. Should packaging reopen?

Support edges: $a_0 \to b_0$ and $a_1 \to b_1$. In both states, τ_1 selects a_1.

State	σ_{τ_2}	After withdrawing a_0
$\mathcal{C}^{(0)}$	b_0	$\mathrm{Desc}^*(\{a_0\}) = \{a_0, b_0\}$; reopen τ_2; τ_1 stays fulfilled on a_1
$\mathcal{C}^{(1)}$	b_1	$\mathrm{Desc}^*(\{a_0\}) = \{a_0, b_0\}$; reopen nothing

Figure 5: Selective correction in the reference model. The two states share every task-level fact, including the edge $\tau_1 \to \tau_2$, the statuses, and the selected artifacts; only the record that τ_2 selects differs. The arrows record acceptance dependencies, not generation ancestry. These states require a prototype extension.

The kernel answers by projecting the invalidated records through the *current selection*, which is the Withdraw rule of Section 3.1:

$$\mathrm{Reopen}(B) = \{\tau : z_\tau = \texttt{fulfilled},\ \sigma_\tau \in \mathrm{Desc}^*(B)\}. \tag{6}$$

Invalidation acts on records and reopening acts on tasks. Mapping every invalidated record to its task would conflate the two and discard independent completions.

Proposition 2 (Exact reopening)**.** *Let $\mathcal{C}$ be valid and let $\mathcal{C}'$ result from $\mathrm{Inv}(B)$ alone. Every record in $\mathrm{Desc}^*(B)$ is not live in $\mathcal{C}'$, and every other record has the same liveness as in $\mathcal{C}$. A fulfilled task becomes open exactly when it belongs to $\mathrm{Reopen}(B)$. Every other fulfilled task keeps its selection, and every outstanding task stays outstanding with a responsible route.*

Proof of Proposition 2. Write $C = \mathrm{Desc}^*(B)$. A record with support in C is reachable from B and so lies in C; hence no record outside C relies on one inside it. Invalidation adds C to W and changes, outside C, no withdrawal mark, record body, revision, or local authority and admission predicate. Induction along a topological order preserves liveness outside C. The selection rule of Inv then reopens exactly $\mathrm{Reopen}(B)$. Canceling a submission can turn submitted into open, but it cannot remove an unaffected selection or end an outstanding responsibility. □

In the example, both states have $\mathrm{Desc}^*(\{a_0\}) = \{a_0, b_0\}$, but their selections differ:

$$\mathrm{Reopen}_{\mathcal{C}^{(0)}}(\{a_0\}) = \{\tau_2\}, \qquad \mathrm{Reopen}_{\mathcal{C}^{(1)}}(\{a_0\}) = \varnothing. \tag{7}$$

In $\mathcal{C}^{(0)}$ the package was accepted on the flawed reconstruction, so packaging reopens. In $\mathcal{C}^{(1)}$ it was accepted on x_1 and stays fulfilled. Reconstruction stays fulfilled on a_1 in both states, and no artifact or check result changes. Withdrawing a_0 does not invalidate a_1 merely because they share a task, and selecting a_1 never redirected the edge into b_0.

Selection versus inference. In $\mathcal{C}^{(0)}$, after a_0 is withdrawn, b_1 is still live. A truth maintenance system labels a node as believed while any justification for it survives (Doyle, 1979), so it would keep τ_2 fulfilled through b_1. The kernel instead reopens τ_2 and waits for a judgment. Here the two records happen to bind the same package, but in general they may bind different artifacts, and delivery returns the artifact named by the selection; an automatic switch would change the delivered object without a judgment authorizing that choice. The rule therefore preserves an independently supported selection, not every task that happens to hold another live record. Recovery takes a fresh acceptance, which may cite a_1 if the requirements permit it, or otherwise a revision first. The records a_0 and b_0 stay withdrawn, and reopening keeps the task's identity and resource charges without promising that the remaining budget can fund recovery. Requiring a judgment is a rule of this protocol, not a consequence of determinism; a deterministic automatic-selection rule would define a different protocol.

Independence is relative to the registered graph and to unchanged local authority and admission conditions. Two records outside each other's closure may still share an evidence source. If that source is found inadequate, identifying every acceptance it affects is a separate authorized judgment, and each affected acceptance must be a seed or be reachable from one. Invalidation propagates the support change it is given; it does not discover undeclared common causes. Exact reopening is therefore a consistency result for those inputs, not a minimum-cost repair.

A.3.3 Correct and minimal withdrawal needs record-level state

The example shows that the right correction depends on which record a task selects. We now show that this information cannot be dropped. Fix a valid state $\mathcal{C}$ and an authorized withdrawal naming eligible records B. A *propagation rule* ϕ maps $(\mathcal{C}, B)$ to a set $\Delta_\phi(\mathcal{C}, B)$ of records to withdraw, with $B \subseteq \Delta_\phi(\mathcal{C}, B)$, and a set $\widehat{O}_\phi$ of fulfilled tasks to reopen. Reopened tasks lose their selections and become open with a route, and the rule may cancel affected submissions as Inv does. It creates or substitutes no acceptance, changes no revision, releases nothing, and leaves record bodies, edges, and resource use unchanged.

Definition 2 (Correct and minimal propagation)**.** *The rule ϕ is* correct *at $(\mathcal{C}, B)$ if every task left fulfilled selects a live record. It is* minimal *at $(\mathcal{C}, B)$ if $\widehat{O}_\phi \subseteq \{\tau : \sigma_\tau \in \mathrm{Desc}^*(B)\}$, so that a task reopens only when its selection relies on a withdrawn record.*

These are the two properties of a correct and minimal build (Mokhov et al., 2018), applied to withdrawal. Minimality concerns reopened tasks; the reference operation additionally withdraws exactly $\mathrm{Desc}^*(B)$, which does not follow from minimality alone. A rule is *task-level* if its choice of $\widehat{O}_\phi$ depends on $\mathcal{C}$ only through the projection $Q(\mathcal{C})$, which keeps task identities, revisions, statuses, selected artifacts and evidence, routes, budgets, resource use, and the task-level edges $\{(\tau(a'), \tau(a)) : (a', a) \in E\}$, together with the tasks that own the records in B. Applying withdrawal marks may still use record identifiers.

Theorem 1 (Task-level rules cannot be both correct and minimal)**.** *No task-level propagation rule is correct and minimal at every valid state and admitted withdrawal.*

Proof. Take $\mathcal{C}^{(0)}$ and $\mathcal{C}^{(1)}$ from Appendix A.3.2 with $B = \{a_0\}$. Both are reachable by valid acceptance sequences, ordering the two package acceptances so that the selected one comes last. They hold the same records and resource charges, select the same artifacts with the same evidence, and project to the same edge $\tau_1 \to \tau_2$, so $Q(\mathcal{C}^{(0)}) = Q(\mathcal{C}^{(1)})$ and a task-level rule returns one set $\widehat{O}$ for both. In $\mathcal{C}^{(0)}$, $\sigma_{\tau_2} = b_0$ is not live once a_0 is withdrawn, so correctness requires $\tau_2 \in \widehat{O}$. In $\mathcal{C}^{(1)}$, $\sigma_{\tau_2} = b_1 \notin \{a_0, b_0\}$, so minimality requires $\tau_2 \notin \widehat{O}$. □

This is Proposition 1 applied to recovery: the task-level pre-state and the withdrawal are the same, and the required updates differ. Two natural task-level rules each give up one property. The rule that

withdraws the named records and reopens nothing is minimal but incorrect at $\mathcal{C}^{(0)}$. The rule that invalidates every record of every task in the task-level forward closure T of the tasks holding records in B, and reopens every fulfilled task in T, is correct. Every record in $\mathrm{Desc}^*(B)$ belongs to a task in T, since record edges project to task edges, and a record owned outside T has no support owned inside T, since otherwise its task would belong to T; liveness outside T is therefore unchanged and no reopened task is left fulfilled. The rule is not minimal: at $\mathcal{C}^{(0)}$ it reopens τ_1, whose selected a_1 lies outside $\mathrm{Desc}^*(B)$, and it withdraws a_1, which the decision did not name.

Proposition 3 (Record-level withdrawal is correct and minimal)**.** *At every valid state and admitted withdrawal,* $\mathrm{Inv}(B)$ *with* $\mathrm{Reopen}(B)$ *as in Equation 6 is correct and minimal.*

Proof. Minimality holds by the definition of $\mathrm{Reopen}(B)$. Correctness is Proposition 2: records outside $\mathrm{Desc}^*(B)$ keep their liveness, and every task left fulfilled selects such a record, which was live in $\mathcal{C}$. □

Together, Theorem 1 and Proposition 3 say that record-level selection and support are necessary and sufficient for withdrawal that is both correct and minimal. Progress is a set of judgments about exact results, each with its own support; it is a property neither of tasks nor of what the harness remembers.

The coarse rule is a conservative alternative within the reference model and the one closest to the prototype, which withdraws at Contract granularity and does not represent several individually addressable acceptances for one task. The state holding both a_0 and a_1 is therefore not a prototype execution. Comparing the two interfaces requires an explicit correspondence or an implementation extension (Appendix B.5).

A.4 Integrity under any harness

Invariant 1 quantifies over every sequence of harnesses. The following proposition shows what that quantification amounts to.

Proposition 4 (A harness reaches the Contract only through admitted operations)**.** *Under Assumption 1, let K be a deterministic implementation of the reference transitions, including its routing choices. Consider two runs from the same valid state $\mathcal{C}_0$, with any harnesses, whose sequences of admitted operations coincide, including payloads, fresh identifiers, and work, check, resource, and scheduling updates. Then their Contracts coincide after every corresponding step, and $\mathcal{C}_t = K^*(\mathcal{C}_0; \Omega_t)$, where Ω_t is the admitted prefix. A computation that produces no admitted operation leaves the Contract unchanged.*

Proof. K maps each state and admitted operation to one next state, and the accept transition builds its record from the pending submission rather than from harness-supplied fields. By Assumption 1 no other write reaches the Contract, and a rejected request changes no authoritative field. Induction over the common sequence gives equal states. Charges and receipts from probes and model calls are not excluded; they belong to the admitted sequence. □

In the step loop of Section 3.2, the harness determines the context, the action, and any request sent to the kernel. Proposition 4 says that this request, once admitted, is their only effect on the Contract, so integrity can be argued over admitted operations without modeling any harness. Determinism alone does not give the invariant, however; the proof below checks the guard and effect of every transition. The invariant is a safety property of the authoritative trace, enforceable by a monitor that mediates every write (Schneider, 2000). Eventual fulfillment is a liveness property, which a rejecting monitor cannot guarantee (Alpern & Schneider, 1985).

Proof of Invariant 1. Induct on the number of transitions, in the style of initialization and preservation (Lamport, 2002). The hypothesis is full validity as defined in Appendix A.1: immutable records, an acyclic graph, Equation 3, routes for outstanding tasks, release certificates, and $0 \le u_\tau \le b_\tau$. It holds initially by assumption. The cases follow Table 1.

Issue, work, check, submission, and scheduling. Issue adds an open task with a route, zero use, and no records, leaving existing records unchanged. Work and check alter only the workspace, receipts, and resource use; accepted objects are immutable, and the reservation guard keeps $u_\tau \le b_\tau$. Submission

binds exact objects to the current revision, making an outstanding task submitted or leaving a fulfilled task's selection in place. Stopping, replacing, escalating, and exhausting change only harness and route fields and keep a route for outstanding work. None of these operations changes support or grants release, and rejections and reads change nothing.

Acceptance. The fresh record is eligible, bound to the current revision, authorized, and admitted, and its complete support is live in the state where the decision takes effect. By Equation 2 it is live, so selecting it preserves Equation 3. Its edges all point into the fresh node, so acyclicity and the liveness of every older record are unchanged. The submission is cleared, and identity, budget, and use are unchanged.

Withdrawal. By Proposition 2, the closure becomes not live, liveness elsewhere is unchanged, every task selected on the closure becomes open and routed, and every other fulfilled task keeps a live selection. Canceled submissions leave either an outstanding task or an unaffected fulfillment. Future acceptance and reuse guards cannot admit a withdrawn record, and traversing the whole graph reaches dependents whose tasks are no longer running.

Revision. All records of the revised task seed the invalidation, so the new revision cannot change a local liveness predicate outside the closure, and clearing selections and submissions cannot change liveness because Equation 2 does not read those fields. The withdrawal argument covers the dependents, and the task becomes open with a route and no selection or submission. The fresh revision identifier prevents old coordinates from becoming current again. Old edges stay fixed, so accepting a replacement under the new revision cannot silently repair an old dependent. Budget and use are preserved.

Release. The same argument removes all records of the released task and everything that relies on them. The task receives its certificate with no selection, submission, or eligible record. Dependent tasks stay open and routed rather than inheriting the release, and since a released task cannot submit, it never supplies prerequisite success.

Every transition therefore preserves validity. Expanding Equation 5 shows that a quiet scope contains only tasks with supported fulfillment or authorized release, and recursive liveness supplies support along every mandatory path. The induction constrains harnesses only through the mediated interface, so it covers every finite prefix of adaptive or randomized version sequences, including prefixes of infinite runs. It implies neither eventual completion nor eventual quiet. □

Atomicity also orders withdrawal against reuse: a use ordered before a withdrawal was authorized in its earlier state, and a use ordered after it fails the liveness guard. The invariant does not recall artifacts already delivered or undo physical effects, and a cached historical certificate cannot authorize a later use without a fresh check.

A.5 Replacing harnesses

Version sequences. Let a run use any sequence of compatible versions $v_0, v_1, \ldots$ (Appendix B.4), whether fixed in advance, chosen from the history, synthesized by an earlier version, or restored by rollback. Every harness step follows the loop of Section 3.2, so a step affects the Contract only through the requests it sends, and by Proposition 4 the Contract is K^* applied to the admitted requests. The proof of Invariant 1 never asks which version emitted an operation. Every such sequence is therefore covered, without a selector, a schedule, or even the compatibility check. Compatibility matters for the other state: it is what lets a successor use the execution state and working material it inherits.

What a version change can reach. A replacement $H \to H'$ changes v, changes m through the migration $\mu_{v \to v'}$, because a new harness may change how memory and context are stored, and may reuse w. It does not write ρ_τ, u_τ, b_τ, A_τ, W_τ, σ_τ, or z_τ, because only K writes them. Two statements of Section 3.2 follow. A change of harness alone does not invalidate a live acceptance, because Equation 2 mentions neither the harness nor its version. An earlier judgment cannot be rebound to a revised artifact, because $x(a)$ is part of the immutable coordinate, so a revised result needs its own acceptance.

Corollary 1 (Rollback moves execution, not authority)**.** *Under Assumption 1, let the controller activate v' from the checkpoint (v_c, m_c, w_c) with Contract $\mathcal{C}_0$, and let Ω be the operations admitted*

during the trial and its handling. Whether the resulting harness state (v^+, m^+, w^+) is the trial's or the restored checkpoint, the Contract is $\mathcal{C}^+ = K^*(\mathcal{C}_0; \Omega)$ and satisfies Equation 3. In particular, restoring the checkpoint neither fulfills a task nor makes a withdrawn record eligible again, and a task whose support was withdrawn during the trial stays open even if m_c records it as done.

Proof. Checkpointing, fencing, restoring, and quarantining write only v, m, and w, which are not authoritative state, so by Assumption 1 they do not write $\mathcal{C}$. Proposition 4 gives $\mathcal{C}^+ = K^*(\mathcal{C}_0; \Omega)$ for either state, and Invariant 1 gives the invariant. A restored m_c can shape only later requests, which pass the same guards; since z_τ and σ_τ are fields of $\mathcal{C}$, no memory of completion can reinstate them. □

Which claim a failed trial disputes. A trial tests one proposition: that v' can carry this task forward from $\mu_{v_c \to v'}(m_c)$ over w_c. It does not test the qualification of v' from fresh state, which was established before activation and is unchanged. It does not test any acceptance made under v', whose subjects and support are unchanged, and it does not test the task's responsibilities. The smallest object a failure can dispute is therefore this *continuation claim*. In the reference semantics, activation records it as a tentative grant, confirmed after a probation window of healthy operation. A trial that stalls during probation leaves the grant unconfirmed and needs no withdrawal. If a confirmed activation later fails, the withdrawal names only the continuation record c. Nothing cites a continuation as support, because parentage is not evidence, so Proposition 2 gives $\mathrm{Desc}^*(\{c\}) = \{c\}$: no other acceptance is withdrawn, and none is restored.

The harness state (v_c, m_c, w_c) is restored as a whole, from a fresh copy of the checkpoint, because m_c was formed over w_c and a memory paired with a different workspace would be inconsistent. The trial's workspace is kept read-only as working material, because what the trial observed may still help the fallback. The three state domains of Equation 8 thus follow three rules: restore execution for consistency, retain observations for information, and never roll back authority. Appendix B.4 gives the procedure and the prototype's coarser rule, which expresses every rollback as a challenge to a broader activation authorization and does not version the workspace.

The corollary does not make execution state portable.

A.6 Self-improvement through the kernel

A revision round is a task whose deliverable is a bundle β, the source of a candidate version, produced by the research parent's own harness, including its generation instructions (Section 3.2). The decisions that Section 3.2 separates live in different places. $\mathrm{AcceptedProposal}(\beta)$ is a Contract fact: the revision task is fulfilled on a live record whose artifact is β. $\mathrm{EligibleToRun}(v)$ is a controller fact: v was built from an accepted bundle and passed qualification from fresh state. $\mathrm{Adopted}(v)$ is again a Contract fact, defined as follows in the reference model.

Definition 3 (Adoption)**.** *For a candidate v and incumbent I, an* Evaluation Contract τ_v^{ev} *fixes the frozen comparison of v with I, namely its tasks, budgets, scoring, and decision rule, and its acceptances are judged by a separate verifier. A* Selection Contract τ_v^{sel} *requires a live acceptance of τ_v^{ev} as support, and its acceptances are judged by the user. Then* $\mathrm{Adopted}(v) \iff z_{\tau_v^{\mathrm{sel}}} =$ `fulfilled`.

Corollary 2 (No version adopts itself, and adoption follows its evidence)**.** *Under Assumption 1, suppose no harness holds the verifier's or the user's authority for these Contracts. Then for every sequence of versions, including v itself and versions generated by v:*

- *(i) if* $\mathrm{Adopted}(v)$ *holds, then $\sigma_{\tau_v^{\mathrm{sel}}}$ is a live user-authorized acceptance whose support includes a live verifier-authorized acceptance of τ_v^{ev}, and no request emitted by a harness creates either;*
- *(ii) if that evaluation acceptance is withdrawn, or τ_v^{ev} is revised or released,* $\mathrm{Adopted}(v)$ *becomes false in the same atomic step, and no task whose selection does not rely on it reopens;*
- *(iii) choosing a research parent writes no Contract field, so exploring from v neither adopts v nor replaces the incumbent.*

Proof. (i) By Invariant 1, fulfillment implies a live selected record. Liveness requires authorization by the user and complete, live support, which includes an evaluation acceptance authorized by the verifier.

A harness holds neither authority, and by Assumption 1 it can neither forge a judgment nor write the Contract directly. (ii) The selected record of τ_v^{sel} lies in Desc^* of the withdrawn evaluation record, so Proposition 2 reopens τ_v^{sel} and nothing whose selection lies outside the closure; revision and release seed the same invalidation. (iii) Parent selection is a controller computation over development results, not an operation in Table 1, so by Proposition 4 it leaves the Contract unchanged. □

The corollary is what licenses unconstrained generation. A generated version may rewrite any part of its harness, including its generation instructions, but whatever it produces is a proposal, and the only route from a proposal to deployment passes through two judgments it cannot make. It also explains why a candidate can serve as research parent without being adopted: the parent role carries no authority, so it needs no evidence beyond the rule that picks it.

The corollary protects who decides adoption, not whether the decision is right. Three conditions lie outside the invariant and are enforced by the controller: the builder, verifier, and data-access rules sit outside the editable agent; the decision rule is fixed before the comparison; and no version reads its own held-out evidence. An accepted evaluation on finitely many tasks also does not order utility on the task distribution. How revisions perform is therefore measured, in Sections 4.3 and 4.4.

A.7 Discussion

Three routes, not one memory. Memory carries past evidence to future attention. The Contract carries current intent to future duty and authority. Verification carries a witness to authoritative status. A long-horizon agent needs all three, and merging them makes each fail in a different way. Proposition 1 shows why the second cannot ride on the first: an authorized event can change what is owed without changing anything the harness observes. Durable workflows make execution persistent but trust the worker to report its own progress (Temporal Technologies, 2026). The kernel makes the responsibility and its progress persistent and treats every harness, including one the agent wrote, as untrusted. The two are complementary; a workflow could host the kernel, provided it keeps the distinctions of Table 2.

Incomplete contracts and residual rights. Acceptance conditions freeze a projection of intent, and no finite projection anticipates every contingency. The Contract is therefore incomplete in the sense of contract theory, and the question becomes who holds the residual right to decide what its terms left open (Grossman & Hart, 1986; Hart & Moore, 1990). The kernel gives that right to the user: only the user revises or releases a task, and harnesses only propose. A harness that meets the letter of the acceptance conditions has fulfilled the task as recorded. If the user meant more, the remedy is a revision, which reopens the task under new requirements without erasing its history or resetting its budget.

Precedents. The record-level machinery has clear ancestors; what is new is their composition around an untrusted, replaceable harness. Truth maintenance systems keep, for each datum, the justifications that support it and retract dependents when a justification is withdrawn (Doyle, 1979; de Kleer, 1986). Equation 2 restricts this idea to a monotone, acyclic, conjunctive graph. The reference model adds an authorization to each record, a status chosen by selection rather than inferred from surviving justifications (Appendix A.3.2), and responsibility, routing, and budget that persist when support is lost. Build systems supply the vocabulary of correctness and minimality (Mokhov et al., 2018), used in Appendix A.3.3, with one deliberate difference. A build system may skip downstream work when a rebuilt input has the same hash, whereas the reference model treats a fresh acceptance of identical bytes as a new record that does not restore old dependents: support is a judgment under a rule and an authority, and the rule or the authority may be what was withdrawn. Commitment protocols model an obligation as a directed relation between a debtor and a creditor, distinguish release from discharge, and let recognized events rather than the debtor's belief advance the commitment (Singh, 1999; Baldoni et al., 2015). The Contract follows this pattern: release ends a duty without asserting success, and a harness's belief that it has finished authorizes nothing. We claim neither compliance with belief-revision postulates nor equivalence to an assumption-based system (Alchourrón et al., 1985); the shared point is only that a conclusion and its grounds must stay connected.

The kernel is also an instance of complete mediation and of an execution monitor for a safety property (Saltzer & Schroeder, 1975; Schneider, 2000). In Kubernetes, replaceable controllers propose changes toward a desired state, and every change passes through the API server, where admission control can reject it before it is persisted (The Kubernetes Authors, 2026b;a). The kernel plays the API server's role for task state, with harnesses as controllers; the difference is that a controller reconciles toward a declared specification, whereas the kernel records discrete authorized acceptances and their support. For self-improvement, DGM replaces the Gödel machine's requirement of a proof of improvement with empirical evaluation and keeps an archive of stepping stones (Zhang et al., 2026). Corollary 2 keeps that openness for generation and adds what an archive does not record: which comparison supports a deployment, who approved it, and what happens to the deployment when that support is withdrawn.

B Implementation Details

OpenCode stores the Contract separately from mutable working state and its Session binding. This appendix describes the interfaces and continuation paths used in the reported benchmark runs, including their differences from the reference semantics.

B.1 State ownership and authority

The execution state contains the current workspace, interaction history, and harness instructions. The task state contains the current requirements, status, current support, responsible route, and resource accounting. The harness can change a candidate or abandon a plan without thereby changing the task's requirements or declaring the task complete.

In OpenCode, model-driven execution occurs in Sessions. A Contract specification persists separately from its execution binding, which contains the instructions, Session association, and lease. This distinction allows execution to stop or move to another Session while the task and its accumulated consumption remain. A lease coordinates execution access; it does not grant permission to accept a result or release a duty.

Table 3 summarizes how each kind of state continues across a restart or a change of harness. Table 4 separates the roles at this boundary. The roles need not correspond to distinct models or human reviewers. An automated check may serve as the verifier, but that role comes from the Contract rather than from its ability to produce a passing result.

Table 3: Continuation rules after a restart or a change of harness.

State	Continuation rule	A harness change alone cannot
Contract	The Contract remains authoritative and changes only through kernel operations; the successor does not reconstruct completion status or resource consumption from its working memory. Persisted acceptance records retain their original bindings, even when they are no longer eligible to support a decision.	Clear responsibilities, reset resource use, rebind an earlier acceptance to a new artifact, or revive withdrawn support.
Working material	Saved probes, observations, and partial implementations can be reused when their task and reference conditions still apply.	Turn a still-useful observation into a still-valid conclusion about a revised result.
Execution state	Context and internal memory formats can depend on the harness. A new version may require migration or reconstruction of this state (Section 3.2).	Overwrite the current Contract with an older checkpoint.

B.2 Scheduling and the implemented continuation paths

The host runner issues the task and initiates a native execution. The harness then chooses actions within that Session. A nonterminal `contract_check` returns a replay outcome and diagnostic

Table 4: Division of control. Roles may share infrastructure, but their authority-bearing operations must remain separately authorized.

Role	Permitted work	Boundary to preserve
Harness	Edit candidates, investigate discrepancies, run checks, and propose changes.	A completion statement or instruction edit cannot itself change acceptance or the responsibility.
Check	Run on the specified candidate and return observations.	A passing result supplies evidence, not authority absent an explicit delegation.
User or verifier	Authorize requirements and the acceptance, withdrawal, or release decisions permitted by its role.	A decision must concern the specified task and candidate and satisfy the applicable admission rule.
Scheduler	Bind work to a harness, stop execution, or route unresolved work.	Execution changes do not erase responsibility or reset accumulated resource use.
kernel	Validate requests and apply authorized changes to task and support state.	No alternative write path may bypass the transition rules; checking authority does not establish evidence truth.

without submitting a handoff, allowing the next model action to continue the investigation. A `contract_report_ready` request instead offers an exact artifact for the delivery path.

Session interruption and process recovery are host scheduling responsibilities. The Contract binding identifies the relevant Session and lease, while the workspace and retained history supply working material.

The final packaging path has a separate limit: the runner sends a candidate error in that phase to its finish-without-delivery path, which keeps the task outstanding but does not resume repair. Five tasks in the selected Sol cohort reach this case before their resource limits; resuming them requires host scheduling.

B.3 Acceptance and withdrawal control

In software reconstruction, the agent uses documentation and black-box probes to infer behavior, edits the implementation, and writes regression tests. A frozen artifact fixes the candidate being checked. The `contract_check` operation returns diagnostics without changing completion status. Failure can therefore guide further work, while a pass leaves the agent free to investigate further.

Acceptance follows a different control path:

1. The harness offers a specific candidate and its evidence. Reporting ready submits a handoff for verification rather than declaring fulfillment.
2. The verifier judges whether the offered evidence satisfies the claim under the admission rule. This judgment must refer to the offered candidate, not a subsequently edited workspace.
3. The kernel checks the decision's authority, task revision, subject, and required support before applying the completion transition. In the prototype this authorized completion is called discharge.
4. Delivery returns the accepted artifact. Later edits remain working state until a further authorized decision permits reliance on them.

A successful tool invocation, a passing check, and a discharged task are thus different outcomes. The kernel controls the transition from evidence to authorized reliance; it does not choose the approach or manufacture stronger evidence.

Native verification checks rebuilding of the captured source and passage of its declared regression replay. The agent writes the replay suite; hidden ProgramBench tests measure behavior beyond it. Kernel acceptance, a scorable delivery, and behavioral score therefore remain separate experimental quantities.

Withdrawal and resources. A challenge proposes that existing support no longer suffices. The authority boundary determines who may accept that challenge; the transition rules determine its consequences. In the prototype, an accepted challenge clears the current handoff and attestation support and routes affected Contract dependents to the user's resolution path. Merely recording a failed test or stopping a Session does not perform this operation.

The reference semantics requires a finer update: invalidate the named acceptance and its dependent records, then reopen tasks according to their current selections.

Resource control follows the same separation. Session rotation preserves accumulated consumption. The reference rules likewise require reopened work to retain its task identity, responsibilities, and original budget accounting. If no budget remains, the task stays outstanding and must be routed to authorized resolution. Exhaustion is neither success nor release, and recovery is not an implicit resource grant.

B.4 VERSION ACTIVATION AND ROLLBACK

This subsection gives the procedure behind the recovery rule of Section 3.2, under which the three kinds of state recover as

$$(v^+, m^+, w^+) = \begin{cases} (v',\ m_{\mathrm{trial}},\ w_{\mathrm{trial}}) & \text{if the trial remains healthy,} \\ (v_c,\ m_c,\ w_c) \text{ with } w_{\mathrm{trial}} \text{ kept read-only} & \text{on rollback,} \end{cases} \qquad (8)$$
$$\mathcal{C}^+ = K^*\big(\mathcal{C}_0;\ \Omega\big) \text{ in both cases,}$$

where K^* applies the kernel to the operations Ω admitted during the trial. Appendix A.5 derives its properties, and Appendix C.2.6 reports the qualification runs.

Versions and compatibility. The activation unit is an immutable harness H: its instructions, context construction, memory and storage code, tools, the execution loop, and build inputs. A harness is compatible, and so belongs to $\mathcal{V}$, when it builds and passes the controller's qualification gate from fresh state; a replacement $H \to H'$ is allowed only when the new harness is compatible. Active artifacts are mounted read-only, and the running harness can export a proposed replacement, which the controller evaluates as a whole. Authorization, independent grading, credentials, and deadline enforcement remain with the external controller, which reaches the existing kernel through an adapter.

Activation. Migration $\mu_{v \to v'}$ depends on both versions, because a new harness may change memory formats. To activate v', the controller checkpoints the quiescent worker's execution state m_c and workspace w_c, stops and positively fences the old harness, and cold-boots a trial in a private state namespace from $\mu_{v_c \to v'}(\mathrm{Copy}(m_c))$ and a copy of w_c. The release pair and its memory lineage are installed together in a durable boot record, so a trial cannot destructively migrate its fallback's state. The trial is healthy while model or tool activity is observed; health does not mean accepted progress or better performance.

Rollback. On loss of observed provider or tool progress, the controller fences the failed worker, quarantines the version, and restores the checkpoint (v_c, m_c, w_c) as a whole from a fresh copy of the compatible checkpoint. Quarantine is a controller policy scoped to the version and its source state lineage, not a withdrawal. The failed trial's workspace is kept read-only as working material. If the restored checkpoint remembers a task as done whose support was withdrawn during the trial, the harness returns but the task stays open. Rollback neither withdraws unrelated acceptances nor restores previously withdrawn ones. Earlier decisions, requests, and the failed trial remain in the task history, an interrupted job retains its original deadline, and external actions already taken are not undone.

The continuation claim. The reference design treats activation as a tentative grant of the continuation claim, that the version can carry the job forward from its migrated state, confirmed after a probation window, so a trial that stalls during probation is simply not confirmed (Appendix A.5). The prototype is coarser: on rollback the controller revokes the failed version's authorization to boot on any state, without challenging the proposal or task acceptances.

B.5 Implementation scope and trust assumptions

Task-level versus record-level control. The prototype defines prerequisites by Contract identifier and revision, retains one current handoff and attestation per Contract, and rejects ordinary further acceptance on a discharged Contract. Challenge targets the current handoff. Older attestations are not individually addressable prerequisite records through this interface.

Selective withdrawal for the state in Appendix A.3.2 therefore requires extension: multiple acceptances for one task, a separate current selection, and withdrawal by acceptance identifier. Treating a_1 as another Contract would still require a mapping from both Contracts to one task. Some revision and release requests with outstanding dependents are also rejected rather than applying the reference model's more permissive invalidation rules. The reference model and prototype therefore expose different withdrawal interfaces.

Authorized judgment versus correct judgment. The verifier-attestation interface checks the caller and decision coordinates but can accept an evidence reference without retrieving or re-evaluating the report. In that mode, the verifier supplies a trusted judgment. Merely naming a report establishes neither its truth nor the independence of its evaluation. A rule requiring authenticated checker results needs an adapter that verifies the report, the evaluated subject, and the required checker authority before admission. Preserving the named cases additionally assumes a sound checker and exact agreement between checked and delivered objects.

Protection and atomicity. The harness must not be able to overwrite the Contract, replace an accepted object, or obtain the verifier's or the user's credentials. All authoritative updates must use the protected interface. Checks of authority and support must apply to the same state in which the decision takes effect, and dependent invalidation must be indivisible from the corresponding task updates. Otherwise a concurrent acceptance could use support that has already been withdrawn. Linearizability expresses this ordering requirement for concurrent operations (Herlihy & Wing, 1990); it does not by itself establish crash durability or reverse prior external effects. The reference invariant assumes the protected boundary.

C Extra Experimental Results

C.1 Evaluation protocol and supplementary results

This section documents the configurations and execution records behind RQ1 to RQ3.

C.1.1 Execution conditions

Runtime and execution environment. Our implementation extends OpenCode, where model-driven execution occurs in Sessions. The server is provided by Google Cloud Platform with an AMD EPYC 7B13 64-core CPU and 512 GB RAM. The host starts execution and controls Session replacement; the Contract persists separately from the Session binding and workspace. In reconstruction, `contract_check` captures the candidate, rebuilds it, and runs its declared regression replay without submitting a handoff. The submission path checks the captured source's build and replay before native acceptance. The agent writes most replay tests from observations; hidden ProgramBench tests evaluate behavior beyond that suite. Appendix B details these interfaces and recovery behavior. Cohort concurrency and resource ceilings are reported below; domain adaptations and the Terminal-Bench CPU/RAM allocation protocol appear in Appendix C.3.

Cohort selection and resource limits. The inventory contains eleven 200-task reports. Selection chooses one whole cohort per model by raw full-score count, then fixed-denominator mean quality, then recorded model cost; it never combines task winners across cohorts. Luna and Sol have six-hour inference ceilings, 1,000 provider turns, and 4,000 tool actions per task. Terra has the same six-hour ceiling without cumulative turn or action caps. None of these configurations has a monetary cap. The selected Terra run uses 32 workers, compared with eight for its earlier Terra configuration, with a different task-order seed.

Scoring. Scores missing from the original evaluation were completed by rerunning the hidden tests on the original submitted artifacts, without new inference; measured failures were not retried, and delivery status is unchanged.

Every report contains the same 200 task identifiers. For a task i with a reported score, let q_i be its exact passed/total fraction; set $\widetilde{q}_i = q_i$ when reported and $\widetilde{q}_i = 0$ otherwise. Let v_i indicate that the benchmark scored the delivery. The raw and invalid-zero means are

$$Q_{\text{raw}} = \frac{1}{200}\sum_{i=1}^{200} \widetilde{q}_i, \qquad Q_{\text{invalid-zero}} = \frac{1}{200}\sum_{i=1}^{200} v_i \widetilde{q}_i.$$

The second quantity is a labeled sensitivity analysis, not a replacement scoring rule. Missing values remain missing in the task-level data; zero is only their contribution to these fixed-denominator summaries.

The external comparison uses the published Vals results cited in Section 4.1. It contains 47 public configurations on the same 200 task identifiers. Thirty-three have reported costs; the other 14 remain in the table but cannot be placed on a cost plot. Exact published resolution flags, rather than rounded task percentages, determine public full-score counts.

Both conflicting cost fields for the public Luna run are retained; they are not two inference runs. Public Fable 5 retains the site's fallback-derived qualification. The plotted frontier lines use only public nondominated entries, with local configurations shown separately.

Table 5 lists the official ProgramBench leaderboard plotted in Figure 3 in the format of its run pages, together with our configurations and the Terra systems of Figure 2.

Table 5: Official ProgramBench leaderboard and our configurations on the 200 tasks. Resolved and Almost Resolved give the percentage of tasks passing all and at least 95% of hidden tests; the mean counts unscored tasks as zero. Every official entry runs mini-SWE-agent. Costs are USD per task as reported by each source; calls count model requests, and NR means not recorded.

System	Resolved (%)	Almost (%)	Mean (%)	Cost/task	Calls
Claude Opus 5 (xhigh)	4.5	37.0	74.68	50.53	51,359
GPT-5.6 Sol (xhigh)	1.0	15.5	69.89	6.08	8,334
GPT 5.5 (xhigh)	0.5	13.5	69.47	8.85	16,320
GPT 5.5 (high)	0.5	5.0	66.51	3.65	8,184
Gemini 3.6 Flash	0.5	4.0	55.66	4.83	29,991
GPT-5.6 Sol	0.5	2.5	57.81	1.00	3,029
Claude Opus 4.8 (xhigh)	0.0	16.5	70.86	21.02	28,977
GLM-5.2	0.0	8.5	64.59	25.36	44,342
Gemini 3.7 Flash	0.0	5.5	61.25	2.04	26,683
Claude Opus 4.7 (xhigh)	0.0	4.5	55.07	10.96	31,877
Gemini 3.5 Flash	0.0	3.0	53.59	5.60	33,181
Claude Opus 4.7	0.0	3.0	50.87	3.81	18,504
Claude Opus 4.6	0.0	2.5	52.08	11.38	51,962
GPT 5.5	0.0	1.5	56.59	1.21	3,723
Claude Sonnet 4.6	0.0	1.0	47.49	26.73	94,373
GPT 5.4	0.0	0.0	37.67	0.33	3,237
Gemini 3.1 Pro	0.0	0.0	36.37	1.51	18,897
Gemini 3 Flash	0.0	0.0	31.75	0.30	16,996
Claude Haiku 4.5	0.0	0.0	29.95	0.80	24,875
GPT 5.4 mini	0.0	0.0	16.39	0.04	3,521
GPT 5 mini	0.0	0.0	16.03	0.03	3,015
PROCONTRACT, GPT-5.6 Sol (xhigh)	2.5	15.0	75.25	3.14	NR
PROCONTRACT, GPT-5.6 Luna (max)	1.5	14.0	70.20	3.26	65,821
mini-SWE-agent, GPT-5.6 Terra (max)[a]	0.5	15.0	72.35	5.38	NR
PROCONTRACT base, GPT-5.6 Terra (max)	1.0	14.0	77.24	13.50	41,278
PROCONTRACT v_2, GPT-5.6 Terra (max)	1.5	14.0	75.72	7.22	35,547

[a] Third-party snapshot; Terra has no official entry.

Table 6 retains the complete third-party comparison population, which Figures 6 and 7 plot against cost as a robustness check on the official leaderboard used in the main text.

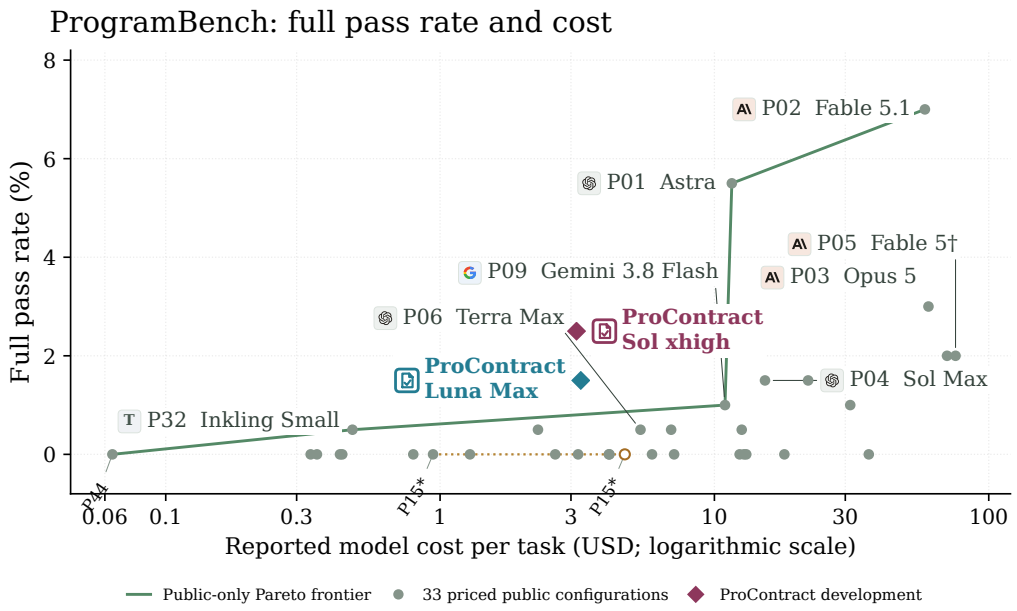

Figure 6: Full pass rate versus reported model cost on the third-party snapshot. The line connects public-only nondominated points among 33 priced public configurations; diamonds show the Luna and Sol ProContract configurations. P15* retains conflicting costs for one public Luna run; † marks fallback-derived Fable 5 data.

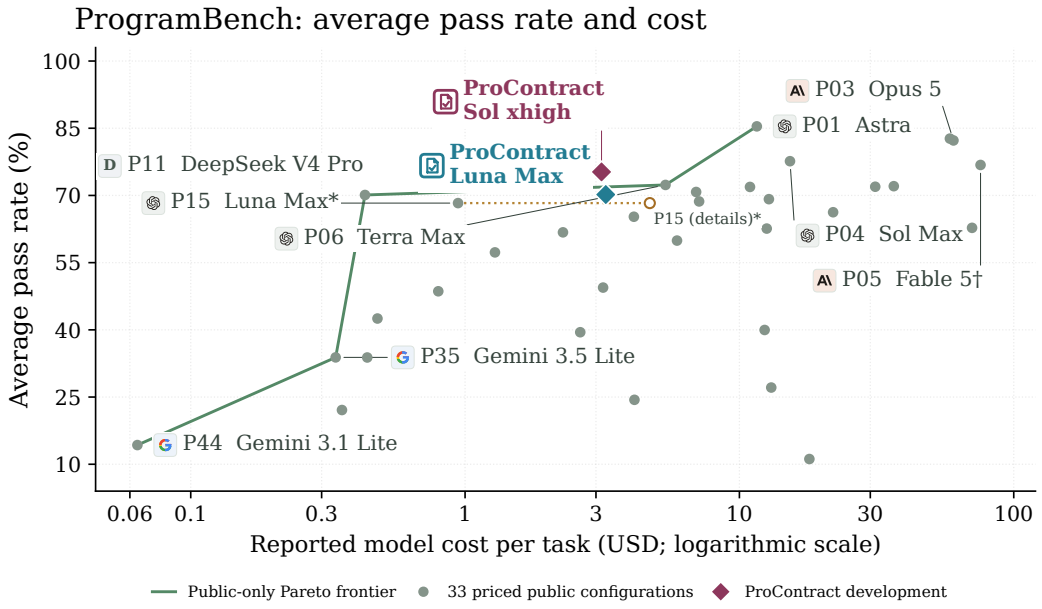

Figure 7: Average pass rate versus reported model cost, using the same records as Figure 6. Fourteen public configurations lack a reported cost and are omitted. Only public entries determine the frontier line.

Table 6: Configuration-level results on 200 tasks: three selected development configurations and 47 published Vals entries. Full gives raw full-score counts. Near counts tasks at $\geq 95\%$. NR means not reported. Costs are recorded USD per task; Terra uses a known usage subtotal and the lower bound of its incomplete-score interval. The inference conditions differ as described in Section 4.1.

ID	Model	Effort	Mean (%)	Full	Near	Cost/task
ProContract: selected development configurations						
L	**GPT-5.6 Luna**	max	70.20	3	28	3.26
T	**GPT-5.6 Terra** (v_2)	max	75.72	3	28	7.22
S	**GPT-5.6 Sol**	xhigh	75.25	5	30	3.14
Vals public leaderboard: mini-SWE-agent; archived snapshot						
P01	GPT-6 Astra	max	85.42	11	100	11.57
P02	Claude Fable 5.1	max	82.69	14	107	58.52
P03	Claude Opus 5	max	82.27	6	83	60.29
P04	GPT-5.6 Sol	max	77.64	3	46	15.29
P05	Claude Fable 5[b]	max	76.80	4	66	75.68
P06	GPT-5.6 Terra	max	72.35	1	30	5.38
P07	Claude Sonnet 5	max	72.07	0	27	36.54
P08	Claude Opus 4.8	max	71.95	2	31	31.27
P09	Gemini 3.8 Flash	high	71.90	2	28	10.93
P10	GPT-5.5	xhigh	70.78	1	19	6.95
P11	DeepSeek V4 Pro 0813	max	70.10	0	27	0.43
P12	Hy4 Preview	NR	69.17	0	29	12.81
P13	Gemini 3.7 Flash	high	68.66	0	19	7.13
P14	GPT-5.4	xhigh	68.60	0	14	NR
P15	GPT-5.6 Luna	max	68.29	0	18	0.94 / 4.71[a]
P16	GLM 5.3	max	66.25	3	33	21.96
P17	Grok 4.5	high	65.25	0	9	4.12
P18	Claude Opus 4.7	max	64.23	0	19	NR
P19	Kimi K3	NR	62.77	4	35	70.48
P20	GLM 5.2	max	62.61	1	19	12.58
P21	GPT-5.4	high	61.76	1	7	2.27
P22	Gemini 3.6 Flash	high	59.93	0	5	5.92
P23	Claude Sonnet 4.6	max	59.10	1	4	NR
P24	DeepSeek V4 Flash 0731	high	57.29	0	17	1.29
P25	Gemini 3.5 Flash	high	56.36	0	3	NR
P26	GPT-5.4 Mini	xhigh	54.38	0	6	NR
P27	GLM 5.1	NR	50.88	0	2	NR
P28	Kimi K2.7 Code	NR	49.44	0	1	3.19
P29	GLM 5.3 Flash	max	48.61	0	14	0.80
P30	Kimi K2.6	NR	47.95	0	1	NR
P31	DeepSeek V4 Pro	max	47.78	0	0	NR
P32	Inkling Small	0.99	42.53	1	1	0.48
P33	Qwen 3.8 Max	NR	39.97	0	21	12.36
P34	Gemini 3.1 Pro Preview	high	39.47	0	2	2.63
P35	Gemini 3.5 Flash Lite	high	33.85	0	0	0.34
P36	Gemini 3 Flash Preview	high	33.83	0	0	0.44
P37	Qwen 3.6 Plus	NR	28.56	0	2	NR
P38	Muse Spark 1.1	xhigh	27.12	0	5	13.06
P39	Claude Haiku 4.5 Thinking	NR	25.42	0	0	NR
P40	Inkling	0.99	24.38	0	0	4.14
P41	Grok 4.3	high	22.11	0	0	0.36
P42	Laguna Xs.2	NR	17.15	0	0	NR
P43	Laguna M.1	NR	15.83	0	0	NR
P44	Gemini 3.1 Flash Lite Preview	high	14.28	0	0	0.06
P45	MiniMax M2.7	NR	13.54	0	0	NR
P46	Qwen 3.8 27B	xhigh	11.17	0	2	17.99
P47	Nemotron 3 Ultra 550B A55B	NR	9.97	0	1	NR

[a] Two conflicting public cost fields for the same Luna run: chart / linked analysis.
[b] Vals describes Fable 5's entire result as fallback-derived; raw published values are shown.

C.1.2 TERRA CONTINUATION AND REVISION EXECUTION

The analysis uses the original 200-task Terra Max / v_2 inference cohort. It joins each task's issue response, persisted Contract, execution events, provider request records, verifier report, and exported archive. The supplementary evaluation supplies the external scores while leaving these execution and delivery records unchanged. The same 200 tasks supply the mechanism observations in RQ1 and the v_2 outcomes and deployed-version checks in RQ2.

State and delivery checks. All 200 tasks have one issue event and retain their issued identity, specification, revision, and deadline. Their provider-side deadlines also agree with the issued deadlines. The 798 Contract commands comprise 200 issues, 200 activations, 199 report-ready operations, 198 discharges, and one escalation. These activations start the Contracts. The tasks declare no prerequisites and contain no withdrawal or release events. The final states are 198 discharged, one escalated (`halite`), and one in verification (`jplot`).

The 435 diagnostic checks comprise 420 passes, nine failures, and six tool errors. Every passing check returns `settled=false`. Among the 199 tasks with a pass, 53 deliver the first-passing subject, 145 deliver another subject with fresh final replay evidence, and one remains undelivered. The remaining task has no passing diagnostic and no delivery. For all 198 delivered tasks, the handoff, replay, attestation, verifier report, native result, and export name the same subject; independently computed report and archive digests match their recorded bindings. In `broot`, `gromacs`, and `svd2rust`, the final subject differs from the last standalone diagnostic check and receives its matching replay during report-ready.

Recovery and resource records. Table 7 lists every v_2 task with model execution in two Sessions. Each retains its original Contract, deadline, and single task attempt. Different admitted provider-peer process IDs, runtime run IDs, and the runner's server-restart phase identify the process boundary. The binaries and solving instructions remain the original v_2. The runner records two process generations for `codesnap` and `gittype`, and three for `datasurgeon`; each task has two Sessions with model execution.

Table 7: Terra v_2 tasks with execution before and after server recovery. Counts are shown as first Session + successor Session. All three tasks reach native delivery. The cost gap is the completed-step cost sum minus the native result's cost field, in USD.

Task	Execution turns	Tool actions	Native cost gap
`codesnap`	$21 + 122$	$32 + 140$	0.0762672
`datasurgeon`	$16 + 85$	$23 + 96$	0.0141108
`gittype`	$12 + 164$	$27 + 172$	0.0137602

Across all 200 tasks, the persisted turn counters equal the number of execution-step starts, and action counters equal tool-call events. The provider records 35,547 sent requests: 35,543 ordinary execution requests and four compaction requests. Three compaction requests lack usage; each occurs in one of the recovered tasks. The native cost gaps in Table 7 compare two recorded monetary fields and remain unreconciled. They are separate from the missing provider usage. The reported Terra model costs use the known provider usage under the Standard-price profile specified above.

The codesnap continuation. The first Session saves reference outputs for no arguments, short and long help, short and long version, and an invalid option in `.probe-initial/`. It writes the probe results; after server recovery, the successor reads them about four minutes later. Eighteen minutes after that, its implementation differs on both help variants and the no-argument response. The successor inspects the byte differences, edits `codesnap.py` twice, and within half a minute matches all six saved exit codes and output streams. It later submits and delivers its accepted artifact. The external source score is 449/571. The trace records each original call, result, timestamp, source-event digest, and final subject identifier.

v_2 identity and code use. The installed executable and all 200 task-local executables match the launch record's binary digest. Each retained initial request contains the complete original solving

instructions. These checks use the deployed Terra artifacts; the generation and role-selection results use the frozen Luna study records.

The observation reader already exists in v_2's parent source. Its 33 Terra calls include 32 successes in 31 tasks, all within a single Session and outside the tasks with compaction or Session recovery. Each returned page matches its stored source bytes, digest, offset, and end-of-file metadata. One `rumdl` call supplies a digest with one mistyped character and is rejected. v_2's source patch changes compaction and its tests. During `jplot`, the one completed compaction retains four diagnostic records with original message and call identifiers, plus an explicit marker for 20 omitted older records. The agent then reads its probe helper, reruns a terminal test, and continues editing; the retained task state is verification.

C.2 Recursive agent revision

RQ2 examines recursive agent revision: how harness code and instructions are generated, evaluated, and assigned subsequent roles. Four studies produce agent revisions, including the executed $v_1 \to v_2$ lineage that connects to the Terra setting of the final configuration. A native Contract accepts each proposal bundle; the host builds and qualifies an executable AgentRevision for evaluation and successor generation. The studies explore distinct development-feedback and source-context settings.

C.2.1 Controller protocol

Algorithm 1 gives the full control flow for separate research-parent and incumbent selection. The controller freezes the incoming parent and incumbent before generating candidates, so proposals use one generation source and replacement decisions use one incumbent baseline within the round. Required controls are evaluated even when no new candidate qualifies, and at most one new finalist enters confirmation. The next parent and incumbent are installed only after the round's decisions are complete. The history retains proposals, failures, evaluations, decisions, and resource use; only permitted development feedback informs later generation.

Algorithm 1 Recursive revision with separate research-parent and incumbent selection

Require: Parent P, incumbent I, permitted experience $\mathcal{E}$, research history $\mathcal{H}$
Require: Fixed generation and evaluation plan, selection rules, research budget B
while the plan and research budget permit another round **do**
  Freeze the incoming versions $P_0 \leftarrow P$, $I_0 \leftarrow I$
  Run P_0 with its generation instructions and $\mathcal{E}$ for the planned proposal Contracts
  For accepted bundles, reconstruct, build, and qualify revisions $\mathcal{C}$
  $D \leftarrow$ evaluate distinct versions $\{P_0, I_0\} \cup \mathcal{C}$ on assigned development tasks
  $P' \leftarrow \textsc{ResearchSelect}(\{P_0, I_0\} \cup \mathcal{C}, D)$, with fallback P_0
  $R \leftarrow \textsc{DevelopmentFinalist}(\mathcal{C}, D; I_0)$; $I' \leftarrow I_0$
  if $R \neq \perp$ **then**
    Evaluate R and I_0 on the assigned confirmation block
    If the fixed adoption rule passes, set $I' \leftarrow R$
  end if
  Append proposals, failures, qualifications, evaluations, decisions, and resource use to $\mathcal{H}$
  Set $\mathcal{E}$ to permitted development feedback from the closed history
  Install the next roles: $P \leftarrow P'$, $I \leftarrow I'$
end while
return $(P, I, \mathcal{H})$

C.2.2 Luna studies: generation and selection

The four performance studies use Luna Max with fixed model parameters. A candidate consists of source edits, solving instructions, and generation instructions that direct the production of later candidates. The generator receives a frozen parent source tree and permitted development observations. Its production Contract requires an agent-revision description, a source patch, and the two instruction files. Native replay, export, and attestation complete the production Contract for the well-formed

proposal bundle. The host then reconstructs the source, builds the executable, and runs qualification checks before development evaluation.

The *research parent* supplies the code and generation instructions for the next candidate; the *incumbent* is the version retained by the study's formal selection rule. In the research-lineage study, the controller selects the research parent using the qualified versions' current development results. Under the Luna-study rule, formal adoption replaces the incumbent and requires complete results and a task-equal mean gain strictly greater than two percentage points over the contemporaneous incumbent on both development and confirmation tasks. For gains Δ_D and Δ_C on development and confirmation, measured in percentage points, this adoption rule is

$$\mathrm{Adopt}(R') \iff \mathrm{Complete}(D, C) \ \wedge \ \Delta_D(R') > 2 \ \wedge \ \Delta_C(R') > 2.$$

Completeness requires valid results for both the candidate and the contemporaneous incumbent.

In research-lineage mode, RESEARCHSELECT compares the incoming parent, incumbent, and all qualified candidates on the current development tasks. It chooses the highest complete, valid task-equal mean; ties retain the incoming parent, then the incumbent, then the earliest predeclared candidate. If no comparison is complete, it retains the parent. This selection uses development results alone. Controls are evaluated even when no new candidate qualifies; identical parent and incumbent bindings share their evaluation. Separately, DEVELOPMENTFINALIST chooses the new candidate with the largest gain exceeding the development margin over the incoming incumbent, with ties resolved by predeclared proposal order. Only this finalist and that incumbent proceed to the assigned confirmation block. Candidate generation uses the incoming parent throughout the round, and the controller installs the next parent and incumbent after closing the round's decisions. Failed proposals and evaluations remain in the research history; only permitted development feedback enters later generation.

The common six-task development set contains `sd`, `eva`, `hexyl`, `direnv`, `xh`, and `pueue`. Development results guide later proposals. The selected finalist in each qualifying round proceeds to its assigned three-task confirmation block. The two consumed blocks contain `ninja`, `pastel`, and `lua` for code revision; and `csview`, `dupl`, and `tig` for research lineage. Confirmation is the final stage of the recorded selection procedure.

Candidate generation and each reconstruction task have an original six-hour deadline, with no cumulative turn, request, action, or monetary ceiling. Individual operations and infrastructure guards remain bounded. These budgets differ from the historical Luna and Sol configurations in RQ3; the later Terra v_2 run also uses a deadline without cumulative count caps. Each study fixes its own development-feedback and source-context inputs to the generator.

C.2.3 COMPLETE INVENTORY

Table 8 includes every generated candidate in the four closed studies. The 108 task executions include candidate runs, controls, and the research-parent retest: 42 for code revision, 18 for feedback context, 12 for source context, and 36 for research lineage. Of these, 105 are valid under their frozen scorers, with zero exact full scores.

The three incomplete candidate groups each contain five valid tasks and one invalid `direnv` execution; their group means remain NA.

C.2.4 A GENERATED VERSION PRODUCES ITS SUCCESSOR

Figure 8 summarizes the two rounds.

Let v_0 denote the initial incumbent, v_1 the first candidate in the research-lineage study, and v_2 its successor. In round 0, v_1 scores 73.00% versus 71.31% for v_0 and becomes the research parent. In round 1, its code and generation instructions execute to produce v_2. The controller selects v_2 as the next research parent. The later Terra configuration executes v_2 on the 200 ProgramBench tasks reported in RQ2.

Round 1 reruns v_0, v_1, and v_2 on the same six development tasks. Their means are 71.19%, 76.78%, and 77.57%. Thus v_2 gains 6.37 points over the initial version and 0.79 points over its retested parent. On confirmation, v_0 scores 92.09% and v_2 scores 91.86%. Each task/version pair runs once, and

Table 8: All nine candidates. Development means are percentages; NA denotes an incomplete six-task group. Confirmation is a gain in percentage points relative to that round's incumbent; "not run" marks candidates that did not qualify for confirmation.

Study / round	Candidate	Valid	Incumbent	Candidate	Confirmation
Code revision / 0	C_1	5/6	72.08	NA	Not run
Code revision / 0	C_2	6/6	72.08	72.85	Not run
Code revision / 1	C_3	6/6	71.36[a]	72.52	Not run
Code revision / 1	C_4	6/6	71.36[a]	74.03	$+0.91$
Feedback context / 0	F_1	5/6	69.45	NA	Not run
Feedback context / 0	F_2	6/6	69.45	69.89	Not run
Source context / 0	S_1	5/6	72.17	NA	Not run
Research lineage / 0	v_1	6/6	71.31	73.00	Not run
Research lineage / 1	v_2	6/6	71.19	77.57	-0.23

[a] The frozen compatibility scorer admits this control's `direnv` result with 850 tests; its stricter validation flag is false. The corresponding candidates have 849 tests. The table uses that study's frozen scoring rule. All development and confirmation results in the research-lineage study are strictly valid with matching task denominators.

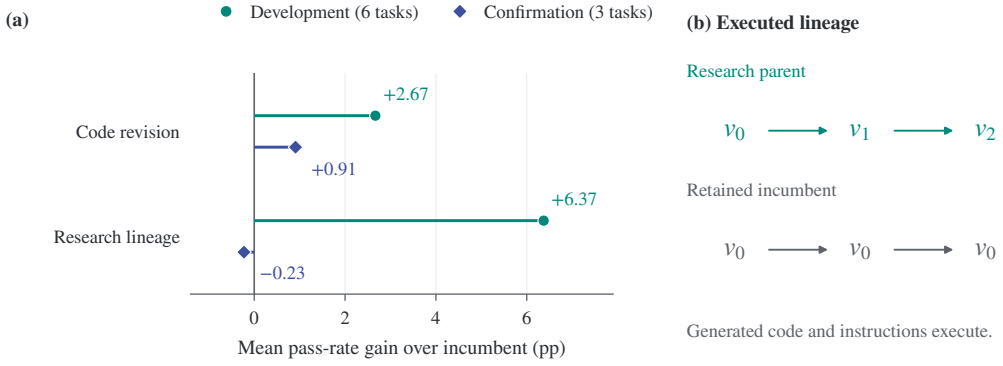

Figure 8: Recursive generation and selection with Luna. (a) Finalist gains over the contemporaneous incumbent on six development and three confirmation tasks, once per version. (b) Generated v_1 supplies the executable and generation instructions that produce v_2; the lower trajectory records the separately retained incumbent.

the scores evaluate the combined code and instruction revision. v_2's largest gains over v_0 are 17.31 points on `direnv` and 10.95 on `eva`; its largest increment over v_1 is 3.76 points on `xh`.

Development-task changes. Table 9 compares all six tasks within that same round. The successor improves over v_0 on every task and over v_1 on four. For `direnv`, passed tests increase from 606 to 739 to 753 out of 849; for `eva`, from 796 to 883 to 896 out of 913. The largest parent-relative gain is on `xh`, from 750 to 794 out of 1,171. The `sd` and `hexyl` regressions remain in the task-equal mean.

Table 9: Contemporaneous development comparison in the recursive lineage. Scores are percentages; gains are percentage points. All task/version pairs are strictly valid with matching task denominators and one execution each.

Task	v_0	Retested v_1	v_2	$v_2 - v_0$	$v_2 - v_1$
`sd`	93.21	99.26	97.65	+4.44	−1.60
`eva`	87.19	96.71	98.14	+10.95	+1.42
`hexyl`	93.49	95.58	94.92	+1.43	−0.66
`direnv`	71.38	87.04	88.69	+17.31	+1.65
`xh`	63.88	64.05	67.81	+3.93	+3.76
`pueue`	18.03	18.03	18.18	+0.16	+0.16

C.2.5 What the generated instructions change

The retained v_2 revision illustrates how a revision spans instructions and code. Its solving instructions prioritize unresolved observations by likely task impact and shared cause. When the cause is sufficiently understood, it calls for a repair followed by replay of the motivating observation and affected neighbors. Otherwise, it calls for a contrast that distinguishes the remaining explanations. The instructions also specify how to recall omitted observations and preserve unavailable or incomplete outcomes as such.

The source patch changes Core Session compaction to collect a bounded set of diagnostic records, retaining invocation identity, input, status, and excerpts with explicit omission markers. When the context budget permits, these records enter the compaction prompt and the resulting summary. This changes the information available for later decisions.

C.2.6 Executable succession and compatible state

The controller qualifies activation and rollback with a scripted provider, real rebuilt headless OpenCode binaries, native Sessions, and tools. Appendix B.4 describes the activation and rollback procedure that the qualification exercises.

The healthy qualification changes context construction, a tool implementation, and the memory schema, rebuilds H_1, and executes it. The next native request contains the successor's instructions and changed context, with the original memory note preserved in the migrated state. In the failure qualification, a revision passes fresh-state checks but damages its inherited memory and stalls during migration. The watchdog restores the previous binary and compatible memory; the restored runtime's next request is observed, while the failed release's attestation is retained. Both cases are scripted-provider mechanism tests, reported separately from the Luna development and confirmation scores.

C.3 Terminal-Bench 2.0

The Terminal-Bench harness derives from the v_2 source with credential-isolation hardening: it keeps v_2's diagnostic compaction and adds domain tools, task prompts, execution controls, budget handling, and tool-transport coordination.

ProContract runs two trials on each of the 89 Terminal-Bench 2.0 tasks (Merrill et al., 2026) with GPT-5.2 at high reasoning effort and a 128,000-token output limit. Task-declared agent timeouts use multiplier 1.0. Success is the benchmark verifier's binary reward, not the kernel's acceptance. The two trials pass 59 and 58 of the 89 tasks; the reported rate is their mean with the Terminal-Bench

interval, pass@1 $\pm$ 1.96 SE over runs (65.7% $\pm$ 1.1). The comparison points in Figure 4 are the public leaderboard entries for GPT-5.2, as reported.

C.4 THE JSONSCHEMA EXAMPLE

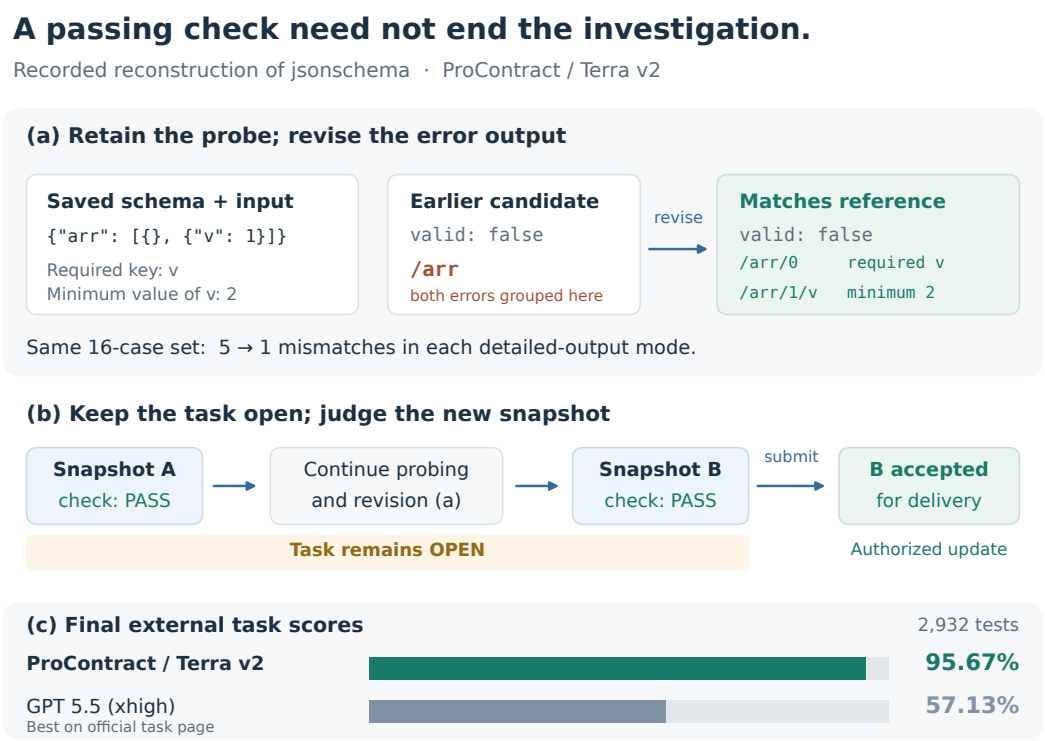

Figure 9: A recorded `jsonschema` reconstruction with PROCONTRACT / Terra v_2. (a) A retained probe exposes incorrect error locations even though the validity flag agrees. On the same 16-case set, each detailed-output mode improves from five mismatches to one. (b) PASS denotes the declared regression replay, not agreement on the separate detailed-output probes. The task stays open until the revised artifact is separately accepted. (c) Final external scores compare our submission with the best run listed on the official task page at the time of writing (ProgramBench, 2026). These are separate runs, not before-and-after scores for the illustrated edits.

Figure 9 follows one reconstruction task in the 200-task Terra v_2 evaluation. At message 237, a native replay passes while the task remains open. The agent continues investigating detailed validation output. On a retained nested-object example, both the reference and candidate report an invalid instance, but the candidate groups two errors at the parent array rather than locating them at the missing property and the failing numeric value. The agent revises the construction of the evaluation tree and replays the same input; the revised output matches the reference in both list and hierarchical modes.

The surrounding probe set contains 16 schema–instance pairs. At message 247, flag output agrees on all 16, while list and hierarchical output each differ on five. After further edits, message 264 reports one remaining mismatch in each detailed-output mode on that same set. The corrected nested-object case is therefore not a claim that every discrepancy was removed. The final candidate receives its own passing check at message 267 and is submitted at message 269; the earlier pass does not close the task or accept the revised object.

The final submission passes 2,805 of 2,932 behavioral tests (95.67%). Its missing evaluation outcomes were completed on the original submitted artifact without new inference. The official task page lists GPT 5.5 (xhigh) at 57.13% as the highest published result for this instance at the time of writing (ProgramBench, 2026). This final-score comparison concerns separate model and harness

runs; the within-run probe sequence above identifies the observed investigation, edits, and replay rather than attributing the full score difference to one edit.

D Extra Related Works

This appendix expands the comparison in Section 5. Tables 10 and 11 compare the state each system retains or edits, the decision that uses it, and the reported implementation or evaluation. ProContract's prototype and reference model have separate rows because record-level recovery is specified and proved in the latter, while the implemented withdrawal operates at Contract level.

The ProContract rows focus on task-bound records governing acceptance, responsibility, and resource accounting; its working material and revision histories also support adapting the approach. When an agent revision is the deliverable, artifact-bound acceptance of the proposal bundle precedes host qualification and the controller's decisions about research use and adoption. The tables below also retain earlier adaptation and execution systems, including OpenHands, which already records cumulative model cost and delegation state.

D.1 Execution state, completion, and recovery

Adaptation and task authority. Reflexion's Actor generates actions, its Evaluator scores trajectories, and its Self-Reflection component produces text that conditions later trials through episodic memory (Shinn et al., 2023). The evaluator may use environment feedback, heuristics, or model judgments. In ProContract's decomposition, the reflection memory guides future actions; an acceptance record has a different role, authorizing reliance on a particular result under the task rules. Feedback can become evidence for acceptance when those rules permit it, but storage as memory does not itself grant that authority.

A Reflexion-style controller could supply adaptive instructions above the kernel by adapting its execution and submission interfaces. The diagnostic loop can alter future behavior; the kernel controls authorized changes to task status. Selecting, revising, or synthesizing instructions can use the former without implicitly changing the latter. The reported ProContract studies exercise designed instruction interventions, retained diagnostics, and generated agent revisions.

ReAct, reusable skills, and memory systems provide ways to guide actions and retain experience (Yao et al., 2023; Wang et al., 2024; Packer et al., 2023; Wei et al., 2026). Feedback-driven revision and tree search appear in Self-Refine and LATS, respectively, while SWE-agent and OpenHands study the interfaces and runtimes through which agents act (Madaan et al., 2023; Zhou et al., 2024; Yang et al., 2024; Wang et al., 2025). Durable workflows, commitment protocols, truth maintenance, and artifact verification supply complementary precedents for retained state and supported decisions (Temporal Technologies, 2026; Singh, 1999; de Kleer, 1986; Doyle, 1979; Torres-Arias et al., 2019). The comparison below focuses on the systems closest to ProContract's execution boundaries.

Table 10: Execution mechanisms and their reported scope. External systems are described from their papers; ProContract's implementation and reference guarantee are distinguished. These are mechanism comparisons, not matched performance results.

System	Retained object	Decision or update mechanism	Reported scope
Reflexion (Shinn et al., 2023)	Trial trajectories and reflective episodic memory	Evaluator feedback drives self-reflection; retained text conditions the Actor's next trial	Behavioral adaptation evaluated on decision-making, reasoning, and coding tasks
SWE-agent (Yang et al., 2024)	Issue-solving trajectory, repository edits, and execution observations	An agent-computer interface organizes navigation, editing, and testing; lint guardrails reject flagged edits	Repository-level repair and interface ablations; submitted patches receive external benchmark evaluation

Table 10 continued

System	Retained object	Decision or update mechanism	Reported scope
OpenHands (Wang et al., 2025)	Event stream, execution state, accumulated model cost, and delegation metadata	Agents map state to actions; a sandbox executes them; delegation routes subtasks to other agents	General software-agent platform with multiple agent implementations and cross-domain evaluations
Ledger (Wang et al., 2026b)	Observed, modified, and attempted repository state	Inform the model; check commands and reuse still-valid results before re-execution	Runtime layer; full SWE-bench Verified evaluation and ablations
Crab (Wu et al., 2026c)	Sandbox process, filesystem, and runtime state	Choose checkpoint granularity from operating-system effects; restore at turn boundaries	Implemented checkpoint/restore runtime; recovery and overhead tests
ECT (Liu, 2026)	Trusted required claims and scoped trace receipts	Return `COMPLETE` after certificate validation and deterministic replay; otherwise continue	Implemented termination gate; synthetic fault and recovery studies
Matrix (Salas, 2026)	Authority, facts, tasks, completion receipts, and dependencies	Verify receipts; retain residual obligations; invalidate affected work and issue replacements	Implemented selective recovery; controlled fixtures with a full-rerun comparison
Execution Lineage (Rosen & Rosen, 2026)	Artifact-producing computations in a dependency DAG	Identity-based replay propagates edits while preserving unaffected artifacts	Controlled policy-memo updates; preservation and consistency measures
PlanFence / ATR (Chen et al., 2026a; Lyu et al., 2026)	Plan-record dependencies / executable action justifications	Revalidate affected dependencies before releasing an action; retain, replan, or block	Implemented protocols; controlled action-validity and systems-cost tests
PROCONTRACT prototype	Task goal, exact candidate, resource history, and responsible route	Nonterminal checks; authorized artifact acceptance; execution rebinding and Contract-level withdrawal	Reconstruction, revision, and domain studies; scripted qualification (Appendix C.2.6)
PROCONTRACT reference model	Selected acceptance records and their support dependencies	Mediated updates preserve live support; withdrawal reopens the dependent recovery set	Conditional state-integrity and selective-recovery proofs; record-level extension pending

Matrix and Execution Lineage already demonstrate dependency-scoped preservation. In Matrix's amendment fixture, selective recovery reissues three tasks versus eighteen under full rerun, with both arms reaching a current state (Salas, 2026). This measures avoided reexecution on that fixture. ProContract's corresponding contribution is the specified relationship between support, task responsibility, harness replacement, and accumulated resources.

Checkpoint recovery addresses a complementary layer. *Resume Means Resume* tests persistence APIs with crash, interrupt, fork, and consume-once probes; *Safe to Resume?* examines inconsistent recovery and stale external dependencies (Khan, 2026; Wu et al., 2026a). A task-level Contract can accompany a sandbox checkpoint without replacing its process-state guarantees.

Evidence requirements also differ. ECT's required claims come from a trusted task contract rather than being authored by the agent during execution (Liu, 2026), and false-success measurements compare explicit completion claims with environment outcomes (Advani, 2026).

D.2 HARNESS IMPROVEMENT AND CANDIDATE SELECTION

Table 11: Improvement mechanisms and selection boundaries. A validation set used to choose candidates is distinct from a final test. SoL-Pi is a project technical report; its proposal pool is distinguished from its retained mechanisms.

System	Editable or retained object	Generation and selection	Closest connection
DGM (Zhang et al., 2026)	Agent source and an archive of generated agents	Generate modifications, evaluate coding ability, and branch from archived versions	Research can use multiple retained versions rather than only one deployed winner
Meta-Harness (Lee et al., 2026)	Harness source, scores, and trajectories	A proposer reads prior candidates and experience through a filesystem	Candidate generation already uses code and traces alongside scores
AHE (Lin et al., 2026)	Editable components, experience, and edit predictions	Pair each edit with a prediction checked against the next round's task outcomes	Makes the intended effect of an edit testable at outcome level
HarnessFix (Chen et al., 2026b)	Step-level traces aligned with harness artifacts	Diagnose a flaw, generate a scoped patch, and validate target improvement with a regression bound	Connects proposed repairs to the runtime steps and artifacts involved
ModularRSI (Wu et al., 2026b)	Five modules, including completion detection	Contrast trajectories; evolve modules; apply program, diff-review, and execution gates; integrate	Completion behavior is editable; frozen harnesses are tested across tasks and models
RSEA (Nguyen et al., 2026)	Instructions, reusable skills, and playbook	Rewrite from trajectories; accept non-regressing working states; update the best state only on strict validation improvement	Separates exploratory working-state updates from retention of the best state
SoL-Pi (Liu et al., 2026)	Harness mechanisms and independent research lineages	Analyze, implement, review, and screen; evaluate frozen candidates on isolated final tasks	Project report: four retained mechanisms from 152 proposals; held-out feedback is withheld from candidate generation
ProContract	Agent source, solving instructions, and generation instructions	Accept proposal bundle; build and qualify; select research parent separately from incumbent adoption	Same Contract lifecycle for tasks and modifications; confirmation participates in selection

The relevant comparison concerns how a modification changes later computation. ADAS, SICA, and STOP explore agent designs and recursive code improvement (Hu et al., 2025; Robeyns et al., 2025; Zelikman et al., 2024). HarnessFix supplies step-to-artifact alignment, and ModularRSI's completion case follows a guard change into later verification actions (Chen et al., 2026b; Wu et al., 2026b). SoL-Pi's proposal pool also includes activation checks, while its reported swarm procedure verifies immutable snapshots before main-agent acceptance (Liu et al., 2026). The comparison distinguishes proposed edits, available mechanisms, and observed execution.

Evaluation protocols determine what improvement means. Wang et al. (2026a) compare harness evolution with parallel sampling, sequential refinement, and instance-specific harness adaptation under comparable feedback and inference budgets, and find inconsistent advantages and limited generalization. HarnessDev further reports dependence on the model executing an evolved harness (Wu et al., 2026d). ModularRSI evaluates a frozen harness under different inference models; SoL-Pi withholds final-test trajectories and results from subsequent candidate generation. Independent evaluation controls adaptive data reuse (Dwork et al., 2015). Our confirmation blocks remain selection data, as specified in Appendix C.2.

Aspire evaluates weight and harness evolution from vague capability goals against hidden tasks. Its controller already maintains private state, a versioned evaluator, candidate eligibility, and explicit retention rules (Wu et al., 2026e). S^3Gym compares raw interaction history, summary memory, and parameter training as ways to turn exploration into improved behavior under held-out evaluation (Shi et al., 2026). These benchmarks test whether proposed changes improve capability; ProContract specifies how accepted task results, responsibility, and resource history persist as the harness changes.

Long-horizon workload design is another axis. EdgeBench provides tasks supporting at least twelve hours of interaction across several domains (Zhu et al., 2026). Such workload structure complements the state and decision boundaries above; a larger time allowance alone does not identify which retained state makes the additional computation useful.

E LIMITATIONS

Integrity is not semantic correctness. PROCONTRACT controls how authoritative task state may change; it does not make the underlying specification or verifier correct. A verifier may accept insufficient evidence, and a finite Contract may omit aspects of the user's intent. The kernel therefore guarantees that accepted progress remains bound to an authorized judgment of a particular artifact under the stated requirements, not that those requirements completely capture the intended task. Stronger specifications, verifiers, and human oversight are complementary to this guarantee.

The reference semantics is finer than the current prototype. The formal model supports multiple acceptance records per task and record-level support dependencies, which enable correct and minimal selective withdrawal. The current implementation instead performs Contract-level withdrawal and uses a coarser rollback rule. Thus, the record-level recovery result is a property of the reference model rather than a production conformance claim. Extending the prototype with record-level selection and withdrawal would test how much of this finer recovery matters in practice; it does not change the harness-independent integrity property that motivates the separation.

Controlled self-modification is not guaranteed self-improvement. Our recursive experiment establishes that a generated harness can execute its own code and generation instructions to produce a successor while evaluation and adoption remain outside the editable agent. It does not establish monotonic improvement, indefinite recursion, or distribution-wide capability growth. Indeed, the generated v_2 improved on the development tasks but did not satisfy the confirmation rule and therefore did not replace the incumbent. This distinction is central to the design: self-modification need not imply self-authorization.

Scope. Our empirical evaluation concentrates on software reconstruction and command-line tasks, where artifacts and verifiers are relatively explicit. Whether the same interfaces suffice for domains with ambiguous objectives, learned or human verifiers, and consequential external actions remains open. The kernel also preserves authoritative task state rather than reversing physical side effects: rollback cannot undo an API call, message, purchase, or other action already taken outside it. Such settings require complementary mechanisms for effect isolation or compensation. Finally, reported model costs exclude verifier compute and the research cost of producing revised harnesses, so they should not be read as complete end-to-end deployment costs.