EXO: AN ARCHITECTURE FOR SELF-IMPROVING SYSTEMS AT RUNTIME

Alexander Krentsel [1 *] **Sylvia Ratnasamy** [1] **Ion Stoica** [1] **Martin Casado** [2] **Ankur Goyal** [3]

[1]University of California, Berkeley [2]Andreessen Horowitz [3]Braintrust

ABSTRACT

Agent harnesses govern how language models act in an environment; designing their policies by hand is difficult, while automated techniques require representative datasets and costly executions. We present *Exo*, an agent runtime architecture for safe learning during ongoing work that collapses the optimization loop into task execution, allowing agents to revise both their harness and their strategies for further improvement. A minimal protected mechanism layer preserves execution history and governs policy activation while the harness evolves through inspectable code changes. On long-running objectives in an open-world game, Exo reduces model cost by 12–53% relative to Codex and Pi in standalone runs and 45–82% in transfer runs, while matching task completion rates. Compared with the same starting policy without the ability to edit policy, Exo reduces held-out model cost by 35–61% on AppWorld and SREGym with comparable correctness, cutting AppWorld's mean task time by 13% and SREGym's median diagnosis time by 35%. On Continual Learning Bench coding tasks, model cost including learning falls by 14%, with both policies resolving every issue.

1 INTRODUCTION

Agents have risen rapidly as a new class of application workload, with wide adoption in developer tools (Anthropic, 2026; OpenAI, 2025a), personal assistants (OpenClaw, 2026; Nous Research, 2026), and enterprise workflows (Pace, 2026; Resolve.ai, 2026; Sierra, 2026). These applications combine a language model with a specialized *harness* whose policy constructs context, provides tools, manages memory, and coordinates actions in an environment. Adapting agents to each domain requires adapting these harness policies: a network operations agent needs different tools, context, and execution policies from a coding agent. Harness engineering makes this specialization possible without changing the model's weights, but developers must decide which policies suit each task.

Building effective harness policies requires substantial development effort. Developers must choose prompts, decide what information to retain and how to assemble context, and select which tools and skills to expose. Making these choices by hand is time-consuming and relies on domain expertise and intuition. Agent-building and optimization tools aim to automate this process on benchmarks: Meta-Harness searches over harness implementations (Lee et al., 2026), GEPA optimizes prompts by reflecting on execution trajectories (Agrawal et al., 2025), AlphaEvolve and SkyDiscover support evaluator-guided program optimization and synthesis (Liu et al., 2026; Novikov et al., 2025), and industry offers similar solutions (Pace Team, 2026). These systems organize development into an external *optimization loop* that proposes candidate harnesses, executes them on benchmarks, and uses results to guide subsequent revisions.

However, external optimization depends on evaluation tasks that may be unavailable and repeated executions that are increasingly costly, while leaving harness policies fixed during subsequent work. In new domains, benchmark tasks and faithful evaluation environments are often unavailable; one study reports that 75% of interviewed teams evaluate their agents without benchmark sets (Pan et al., 2025). Even when benchmarks exist, the trend toward longer task horizons increases the cost of evaluating candidates through task completion, with some reported executions already lasting days or weeks (OpenAI, 2025b; Lin, 2026; Carlini, 2026). Benchmarks can also become outdated as the environment changes. Finally, the resulting harness policies remain fixed during execution, even when an agent's runtime experience may expose a possible gap. Taken together, these challenges can make external agent-building techniques impractical in real-world settings or limit their effectiveness.

In this work, we propose an alternative approach to harness adaptation: equip a minimal agent to inspect and edit its own harness, then let it use experience from ongoing work to revise and adapt itself at runtime. As we explain in §3.1, two properties enable this departure from traditional external adaptive control: the agent can inspect and reason about the code implementing its own harness, and it can implement policy revisions with the same code-editing tools it uses

*Corresponding author: akrentsel@berkeley.edu

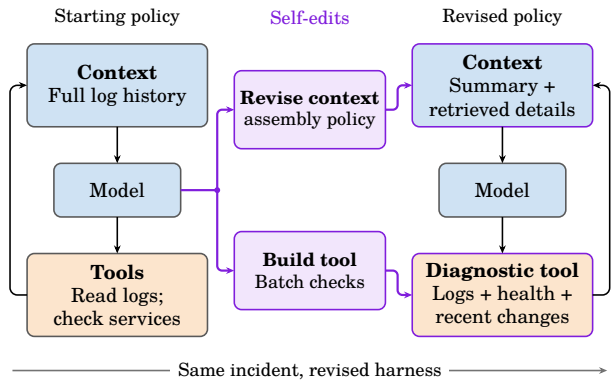


Figure 1. Illustrative harness adaptation during a service incident. The agent builds a diagnostic tool and revises context assembly while continuing the same task. Model weights remain unchanged.

for ordinary task actions. This enables us to *collapse the optimization loop*: the agent chooses both task actions and changes to how it acts (Fig. 1). Adaptation need not wait for retrospective review of completed execution traces: with self-editing available as an explicit action, the agent can explore alternative behaviors, assess their effects, and decide whether to incorporate them into its harness during the task. This design allows for the very policy the agent uses to govern its action to evolve at runtime with experience, including to improve its ability to improve itself.

However, enabling agents to edit themselves introduces two core systems challenges. First, the resulting *fate-sharing* between the optimizer and the running system means that a faulty edit can disable both the agent and the logic needed to diagnose and repair it. Second, self-edits can erase unprotected state – execution history, memory, and task artifacts – destroying accumulated progress and the record of past actions, successes, and failures. An agent cannot repair itself if it cannot execute, nor learn from experience it has lost.

To address these, we present Exo, an agent runtime architecture for learning and adaptation during ongoing work, with mechanisms to support safe evolution of an agent's harness policy. Our key idea is to partition the agent into an evolvable policy layer and a minimal set of protected mechanisms necessary for safe operation. As we show in §4, these mechanisms preserve execution history and prior state across policy revisions and keep recovery authority outside the editable layer. Inspired by operating systems work on separating policy from mechanism, particularly exokernels (Engler et al., 1995), we expose primitive interfaces that leave as much policy as possible in the evolvable layer, including how the agent uses its experience and improves itself. Exo's architecture makes learning observable and adaptable while limiting the damage a faulty revision can cause. The agent can examine its execution history, including outcomes, model costs, and cache use, to identify improvements grounded in the environment where it actually works. It can use that evidence to revise both its task policy and its optimization strategy. Throughout this process, protected history and snapshots preserve the experience and recovery points needed to continue learning after a failed revision, while an independent guardian controls when revised harness policies take effect (§4). The resulting adaptations are interpretable: they appear as explicit changes to harness code that developers can inspect.

We evaluate two benefits of runtime learning: improving efficiency within complex tasks and transferring the resulting specialization to similar tasks. We study long-task adaptation in RuneScape, a dynamic, open-world multiplayer role-playing game, across combat, crafting, and currency objectives. In standalone runs, Exo completes combat and crafting at 12–53% lower model cost than Codex and Pi and earns 44–125% more currency per model dollar. In combat transfer runs on fresh accounts, the cost reduction is 72–82%. We also compare Exo to the same starting policy without the ability to edit policy on AppWorld, SREGym, and Continual Learning Bench (CLB) coding tasks (Asawa et al., 2026). On AppWorld, Exo reduces held-out cost by 61% at the same observed 96.9% pass rate and total cost including learning by 43%. On 20 held-out SREGym incidents, it reduces median diagnosis time and test cost by 35%, and model rounds by 38%. On CLB, it uses 15% fewer shell steps and 14% less model cost including reflection, with matching issue resolution rates across five runs.

In this work, we make three contributions:

- We propose runtime harness self-editing and identify the properties that allow the running agent to revise both its task policy and its improvement strategy, collapsing the optimization loop.
- We identify the challenges of runtime self-editing and address them through an architecture that separates evolvable policy from protected mechanisms for preserving state and controlling policy changes.
- We implement and release Exo as open-source software (Exo Contributors, 2026), and evaluate it across open-world objectives and three graded benchmarks, demonstrating efficiency gains during ongoing work and reuse of learned policies on subsequent tasks.

2 Background

We first define agents and their policy, then describe the challenges of adapting policies to real-world workloads.

2.1 Agents and Harnesses

An *agent* couples language-model inference to an environment through a repeated interaction loop (Yao et al., 2023). We view the *harness* as the scaffolding surrounding the model that (1) assembles the model's input and (2) executes and observes actions on the *environment* (Anthropic, 2025), which is most often a computing sandbox. A *task* is a unit of work with an objective and completion criteria, which may be implicit. Task granularity varies: an SRE agent might manage one software rollout or a month's availability SLO across all rollouts.

Harness policy. We use *policy* to refer to all of the programmable rules and procedures governing how an agent uses its model, state, and environment. These include tool implementations and selection; prompting; context assembly and compaction; memory and skill indexing and retrieval; and external integrations, such as service adapters and task scheduling. These choices are encoded in harness code, prompts, configuration, and reusable skills. Policy also includes the strategy for deciding when and how to attempt further improvements.

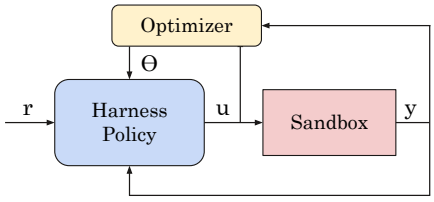
Though tools in policy and agent-written code in the environment are similar in spirit (in what they can accomplish), they differ in how they are managed by the agent. Policy-level tools are often wired directly into context assembly, whereas code in the environment must be remembered and managed entirely by model reasoning.

2.2 Challenges in Harness Adaptation

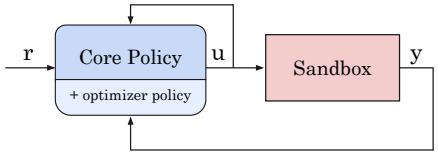
Automated optimization methods develop harness policies through an external loop that proposes candidates, executes them on evaluation tasks, and uses the results to guide revisions (Agrawal et al., 2025; Lee et al., 2026). Three limitations motivate a different approach.

Evaluation datasets are difficult to obtain and maintain. New domains often lack representative task sets and faithful evaluation environments. In a study of production agents, 75% of interviewed teams evaluated without benchmark sets (Pan et al., 2025). Constructing a benchmark requires capturing relevant tasks, expected outcomes, and the environment in which actions take effect. Even then, changing services, interfaces, and user requirements can make those tasks and environments outdated (Krentsel et al., 2026). An optimizer may consequently improve performance against a proxy that no longer reflects reality.

Longer tasks increase evaluation cost. Evaluated task horizons are increasing (METR, 2026), with software-building and theorem-proving projects already spanning weeks (Carlini, 2026; OpenAI, 2026b). As executions lengthen, evaluating successive candidate policies through complete task runs becomes more expensive. Waiting for task completion also delays feedback, even when an inefficient procedure or recurring failure becomes apparent much earlier.



(a) Classical "adaptive control" with optimizer and controller.



(b) Collapsed optimizer + controller (Exo's approach).

Figure 2. Collapsing an adaptive optimization loop. In (a), a separate optimizer maps execution feedback to harness updates θ. In (b), the agent chooses task and revision actions through the same action interface u.

Fixed harness policies limit runtime adaptation. Even a policy optimized for a representative workload can encounter new patterns during execution. A fixed harness can react to observations and retain memories, but its procedures for constructing context and executing actions remain unchanged. For example, an agent may recognize that most diagnostic output is irrelevant while its context-construction policy continues to include it. Without a path from that observation to a policy revision, the same inefficiency can recur throughout the task.

3 Approach

To address the challenges raised in §2, we propose an alternate method of policy creation: runtime harness adaptation, where the running agent uses experience from ongoing work to revise the policy governing that work. This brings task actions and policy revisions into the same decision process.

3.1 Collapsing the Optimization Loop

The external optimization systems described in §2.2 can be viewed through the lens of control theory as adaptive control loops (Astrom & Murray, 2008). In a traditional adaptive control model, shown in Fig. 2(a), the controller (harness policy) maps an objective r and observations of the environment y to task actions u (tool calls). An optimizer then observes the same execution and produces an update θ to the controller policy.

Harnesses have two special properties that allow us to bring the optimizer into the core policy itself. First, a harness revision can be realized in the same action domain as ordinary task work, which we refer to as *optimizer symmetry*. More precisely, the update θ can be produced by a sequence of actions in $\mathcal{U}^*$: the agent can inspect source, edit files, run tests, and request activation using tools already exposed through u. Revision actions therefore become choices available to the core policy alongside actions that act directly on the task.

Second, the harness is *operationally homoiconic*: its policy is code, and that code is represented as files the agent can observe and manipulate with its code-editing tools. The object being optimized is consequently available as data in the optimizer's environment. Together, same-domain actions and editable policy representation allow the optimizer's proposal logic to move inside the harness, producing the collapsed loop in Fig. 2(b).

Because the policy governing when and how to revise the harness is itself editable, the agent can revise both its task behavior and its strategy for further improvement. Each successor can adopt a new policy while retaining the ability to revise itself further, independent of model retraining.

3.2 State and Replacement Contracts

Bringing revision into ongoing execution creates two systems challenges. First, it introduces *fate-sharing* between core logic and optimizer logic, meaning a faulty edit can disable both the agent and the logic it would use to diagnose and repair itself. Second, edits to *unprotected state* can erase accumulated progress and execution evidence, including records of failed revisions.

To address these two critical failure modes, we turn to system architecture. We first model the system's runtime state and transitions, then specify the properties the architecture must enforce as the agent revises its policy.

State and transitions. We call the replaceable program that implements the agent's harness policy the *executor*. We take runtime state at step t to be s_t, and let a protected mechanism K apply the executor's chosen operation a_t:

$$s_t = (d_t, e_t, \pi_t), \tag{1}$$

$$s_{t+1} = K(s_t, a_t). \tag{2}$$

Here d_t contains durable data, including history, memory, and artifacts; e_t contains sandbox files, dependencies, and processes; and π_t identifies the active executor implementation and configuration. Operations may update data, act in the sandbox, or request replacement. In order to stay protected, the storage and replacement rules in K must remain outside the executor's editing authority.

Storage contract. Let $H(d)$ denote canonical execution history: the record of messages, tool calls and results, and policy-replacement events. The storage contract requires this history to be append-only:

$$H(d_t) \preceq H(d_{t+1}), \tag{3}$$

where $\preceq$ denotes the prefix relation. Existing records cannot be overwritten or deleted. Memory and task artifacts may have mutable current values, but updates must retain prior versions for recovery. These rules preserve evidence without prescribing how the policy uses it.

Replacement contract. Editing source produces an arbitrary candidate π', which may have any sort of behavior including crashing; activating this as the active policy must therefore require a check that lies outside of the policy itself. We call this mechanism the *guardian* $G(s_t, \pi')$, which evaluates to $\{0, 1\}$. Thus, for a replacement request,

$$\pi_{t+1} = \begin{cases} \pi', & \text{if } G(s_t, \pi') = 1, \\ \pi_t, & \text{otherwise.} \end{cases} \tag{4}$$

Policy replacement preserves durable data and leaves the sandbox unchanged, while appending replacement events to history; the acceptance decision may inspect both.

Taken together, these contracts enable the agent to revise its policy with maximal flexibility while ensuring that edits cannot erase protected history or bypass replacement checks. Notably, this protection holds regardless of how the agent chooses its edits, but does not on its own guarantee improved behavior or continued progress. We give the preservation proof and discuss these guarantees' assumptions and limits in §A.

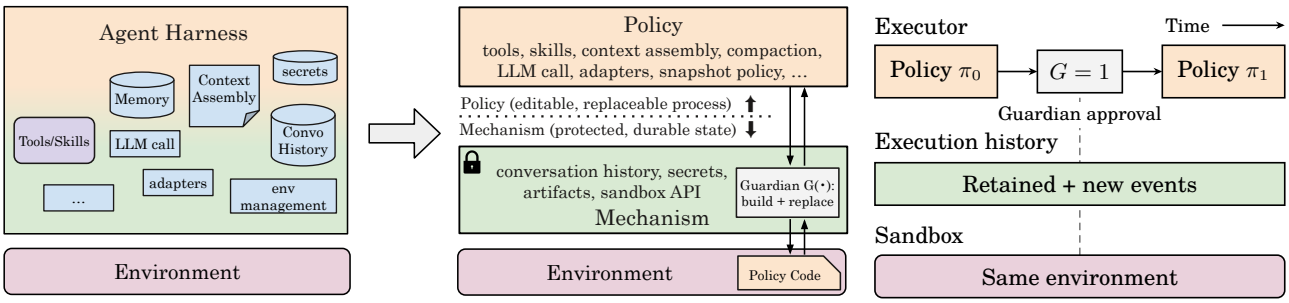
4 DESIGN

The goal of Exo's architecture is to implement the state and replacement contracts in §3.2 to provide safety, while enabling the agent to self-edit at runtime with as much freedom as possible to change how it works. We now describe our architectural design decisions to enable this.

4.1 Partitioning Policy and Mechanism

We consider agent architecture through the lens of a computing system. The traditional view of harnesses combines two different concerns: the infrastructure required to serve an agent, including durable history, privileged resources, and sandboxes; and the policies governing how the agent thinks and acts, including prompt assembly, compaction, memory, and tools. This makes self-evolution difficult to do safely, as the evolvable policy is not distinct from protected resources.

Our key guiding design decision is to partition the harness into two distinct layers: a protected mechanism layer and an editable policy layer, akin to the kernel and userspace



(a) Policy–mechanism partition. **(b)** State across replacement.

Figure 3. Exo separates editable policy from protected mechanisms (a). Policy source is exposed for self-editing, with executor activation controlled by a guardian. Across an accepted executor replacement (b), history is retained and extended, and the sandbox can remain in place. State and replacement contracts are given in §3.2.

split in operating system (OS) design. To determine what functionality should go in each, we draw on inspiration from Exokernel (Engler et al., 1995), aiming to isolate the minimal set of functionality needed for our safety properties, and to expose it at the lowest level of abstraction to maximize the space of behaviors that can be expressed above it.

We show our chosen partition in Fig. 3(a). Below the boundary, Exo protects the state and mechanisms needed to preserve continuity across self-edits. These are the canonical execution history, which is exposed through an append-only queryable interface annotated with costs, along with a versioned artifact store that can hold arbitrary data such as memory, skills, or sandbox snapshots. Finally, Exo provides the protected mechanisms for credential storage, sandbox API, and the independent guardian mechanism that controls activation of replacement executors.

All other harness functionality is expressed in the policy layer, which we call the *executor*. These include tool implementations and selection; context assembly and compaction; prompting; memory and skill indexing and retrieval; and external integrations, including service adapters and task scheduling. For example, the protected mechanism preserves execution history, while the policy decides which parts of that history are included in the prompt sent to the model. The agent can revise any policy code above the protected boundary, including its tools, model calls, compaction, and the procedures governing when and how it attempts further improvements.

4.2 Enabling Runtime Self-Editing

To enable runtime self-editing on this architecture, we take two steps:
(1) Mount the executor code in the sandbox, allowing the agent to inspect and edit it with its existing tools.
(2) Provide a tool to request executor replacement through a protected guardian, which builds the candidate and activates it only if it passes a guardian function G.
The agent can thus inspect and edit its own implementation during task execution, then request activation of the revised policy (Fig. 3(b)). The guardian runs independently of the executor, and replacement requests and outcomes are recorded in the event history. Our default guardian checks build success; richer implementations could evaluate candidate behavior over a longer observation period (§7).

Independent rollback. Exo's split also enables independent rollback of executor policy, durable agent state, and environment sandboxes. Executor rollback restores an earlier code version and reactivates it through the guardian. Append-only history supports forking from an earlier event, while versioned artifacts retain earlier state for recovery. The sandbox API separately supports snapshotting and restoring the task environment. These operations preserve the original execution record, allowing the agent to retain evidence of what it tried even after a rollback. Snapshot scope and limits on reversing external effects are discussed in §A.

4.3 Internal Legibility

To guide self-improvement, an agent needs to inspect both its past behavior and the costs it incurred. Exo exposes execution history for retrieval and analysis, annotating model-message events with cost and token counts, including cached tokens. This makes earlier execution available for inspection even when it is no longer included in the model's context. Policy code can surface these annotations alongside the messages it retrieves to assemble context, or through tools that summarize execution statistics. Our implementation provides a reporting tool covering spending, cached input share, repeated tool calls, and context composition. For example, the agent can inspect whether a context revision reduces input tokens while preserving cache reuse.

Implementation and deployment. We have implemented and released Exo as open-source software (Exo Contributors,

2026). The core runtime architecture, including protected mechanisms, totaled ∼20,000 lines of code, and the base starting policy comprises ∼1000 lines. The runtime is deployed in production, having served O(10k) diverse tasks over O(5m) messages.

5 EVALUATION

We evaluate two facets of Exo's runtime learning: (1) whether specialization improves efficiency within an ongoing task, and (2) whether the resulting tools and policies remain useful on subsequent tasks. We first compare Exo with existing harnesses on long-running open-world objectives, then examine how specialization develops and transfers. Finally, we evaluate efficiency and correctness across graded application, infrastructure, and coding tasks.

5.1 Experimental Setup

Comparisons. Our open-world experiments compare Exo, Codex (OpenAI, 2026a), and Pi (Zechner, 2026) using `gpt-6.1-sol` at medium reasoning effort across different objectives. For AppWorld, SREGym, and Continual Learning Bench (CLB) coding, we use `gpt-5.6-sol` and compare Exo to the same starting policy without the ability to edit policy. This control includes standard tools for coding, shell access, subtask planning, and memory and is labeled *Control* in our figures. We describe each workload's task sequence and feedback below.

Metrics. We report task outcomes alongside model cost and time. For finite game objectives, we measure cost to completion and progress per model dollar over a fixed window. All costs use common token pricing that accounts for caching. For graded tasks, we report correctness and execution efficiency, distinguishing cost on held-out tasks from total cost including learning and reflection. A *standalone run* starts without retained agent state; for game objectives, a *transfer run* repeats the objective on a fresh account while retaining agent state.

5.2 Specialization During Open-World Execution

We first examine runtime specialization in a long-running task in an emulated environment based on RuneScape, an open-world multiplayer role-playing game (Jagex, 2026). Agents act through the common RS-SDK interface (Bittker, 2026) to pursue objectives, starting from new accounts. We select three distinct objectives: training all combat stats to their maximum; accumulating currency through any available strategy; and directly crafting a challenging item, the adamant platebody, which requires resource gathering and training, without buying the finished item. These objectives require hours of coordinated action across a very large open world with a dynamic environment of monsters and shared resources. Combat runs were performed on a public server with 1,200-1,500 other players and bots. The other tasks reported were run on a shared private server due to public server overload. All agents run concurrently in the same game world.

Exo completes the finite objectives at lower model cost than Codex and Pi, and earns more currency per model dollar. Fig. 4 compares standalone and transfer outcomes; Tab. A1 provides a tabular view of these outcomes. In standalone combat, Exo spends $3.71, versus $7.26 for Codex and $7.95 for Pi; in crafting, it spends $2.99, versus $3.90 and $3.41. Together these yield 12–53% lower completion cost. For money-making, Exo earns approximately 332k coins per dollar, versus 231k and 148k, an improvement of 44–125%. All arms finish the finite objectives.

5.2.1 *Looking Inside Runtime Specialization*

Exo shows an early investment in specialization followed by a lower rate of model spending during the same task. Fig. 5(a) shows progress trajectories across agents for the combat task. At ten minutes, Exo has spent approximately $0.31, versus $0.12 for Codex and $0.16 for Pi (Fig. 5(a)). Much of the early work for Exo builds and revises a reusable combat tool and edits context assembly policy, while Codex and Pi make progress towards their goals with comparatively less reasoning. However, as a result of these changes, Exo quickly optimizes its spend rate, with Pi's cumulative spending overtaking Exo at 25 minutes, and Codex's at 86 minutes. We call this early expense the "specialization cost", where self-improvement pays an early premium for improved later performance within the standalone execution.

Exo's specialization appears as concrete changes to its tools, action procedures, and context policy. Fig. 5(c) highlights three of the sixteen self-edits Exo made at runtime. At A, Exo creates `train_combat`, which batches melee actions and reports experience gained and observations. After repeated failures to attack a target, it revises the tool at B to skip stalled targets and retreat earlier. At C, it builds `combat_cycle`, which selects the lowest uncapped melee skill, trains for 180 seconds, collects nearby resources, and stops on low health. Later, it changes context assembly to retain only four earlier turn summaries, leaving older strategy in durable memory. These edits replace repeated model decisions with reusable procedures and change how the agent presents its experience to the model, which result in meaningfully more efficient agent behavior.

Specialization may not repay its cost when work offers little reuse. We selected one additional objective representative of the least repetitive class of task: completing as many quests as possible. Each quest is completed once and requires largely distinct steps, items, and locations. Across three runs per arm, Exo averages $0.28 per quest point, ver-

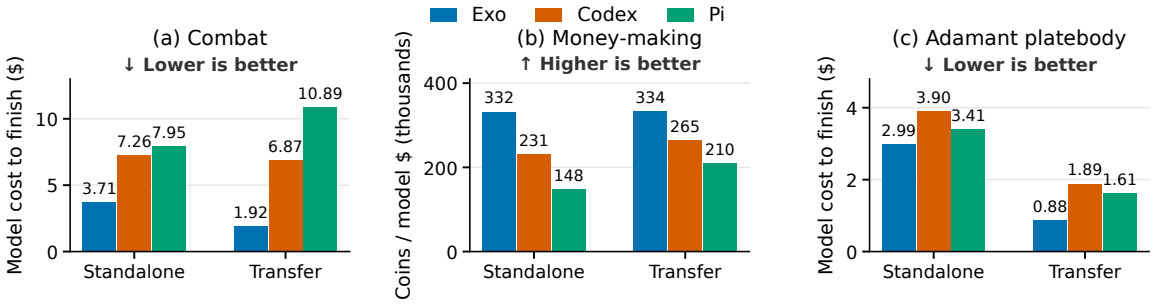


Figure 4. Open-world outcomes in standalone and transfer runs. Combat and crafting report model cost to finish (lower is better); money-making reports coins per model dollar near eight active hours (higher is better).

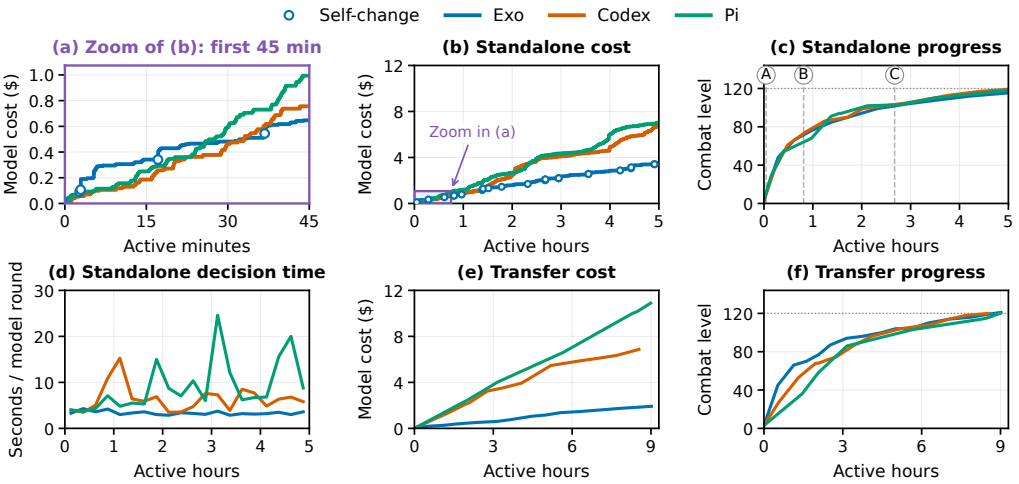


Figure 5. Combat specialization and reuse. (a) First 45 active minutes, magnifying the purple box in (b). Circles in (a,b) mark Exo's tool changes and memory updates. (b–d) Standalone cost, progress, and decision time through five active hours; A–C identify tool changes discussed in the text. (e,f) Transfer cost and progress on fresh accounts with prior agent state retained; A spike in public-server usage lengthens transfer runs. Cost in (a,b) steps at model calls. Decision time is the median time from input to the next model action in each 15-minute window, including provider and network delay.

sus $0.24 for Codex and $0.21 for Pi. Exo builds reusable dialogue tools, but the quest-specific work leaves little opportunity to amortize specialization cost. Thus other harnesses without self-reflection or improvement operate more cheaply for such a task.

5.3 Transfer of the Specialized Harness

The tools and policies learned during standalone runs remain useful when the agent repeats the objectives on fresh accounts. All three arms retain their history and sandbox state; Exo also carries its self-modifications. As shown in Fig. 4, Exo's combat cost falls from $3.71 to $1.92, versus transfer costs of $6.87 for Codex and $10.89 for Pi. Its crafting cost falls from $2.99 to $0.88, versus $1.89 and $1.61. Exo thus completes these transfer objectives at 45–82% lower model cost than the other harnesses. Its own efficiency improves from the standalone to transfer run as it has paid the upfront learning cost. Its money-making efficiency remains approximately unchanged, as it found the close-to-optimal strategy early in the standalone run.

In combat transfer, Exo reuses its specialized harness without further self-changes. The low spending rate in Fig. 5(e) reflects the application of tools and context changes acquired during the preceding run. All arms retain their experience, but their resulting policies use that experience differently: Exo can invoke its learned action procedures directly. As Fig. 5(f) shows, all three arms reach the combat target after approximately nine active hours, longer than the standalone run, while Exo maintains its model-cost advantage. This experiment demonstrates reuse on a repeated objective under changed execution conditions.

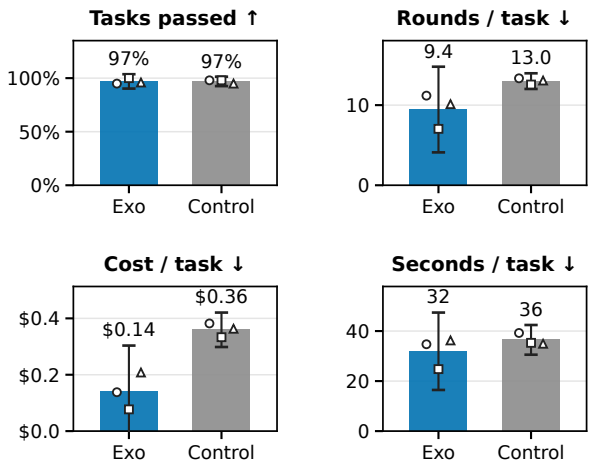


Figure 6. AppWorld held-out outcomes across three seeds. Bars are means, symbols show seeds, and error bars are 95% t-intervals. Control uses the same starting policy without the ability to edit policy.

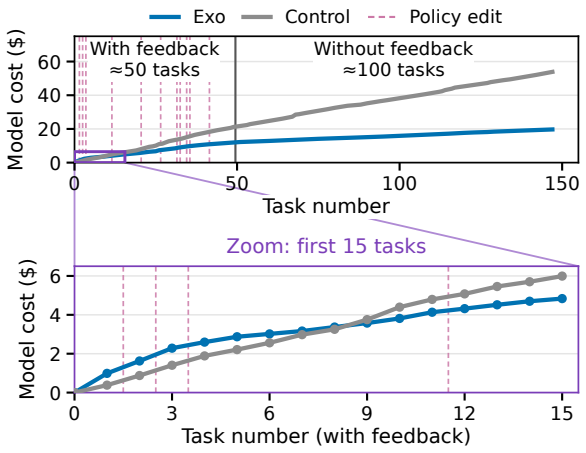


Figure 7. AppWorld: cumulative model cost including reflection over a full run (top), with the purple box enlarged below. Dashed lines mark Exo's policy edits. Exo's cumulative cost falls below Control's at task 9.

5.4 Learning Across Graded Tasks

We next evaluate whether harness adaptation improves efficiency across application, infrastructure, and coding tasks. For all three benchmarks, we compare against the same starting policy without the ability to edit policy. We restrict network access on all arms to prevent cheating. All arms receive the same correctness feedback after learning tasks. AppWorld and SREGym then use held-out tasks without feedback; CLB measures learning across a fixed issue sequence. Tab. 1 summarizes the primary outcomes.

5.4.1 AppWorld

AppWorld (Trivedi et al., 2024) presents user requests spanning nine everyday applications, such as computing payments made to roommates. Exo learns on one variant of each of 50 scenarios, with grader feedback and reflection after each task, then executes 100 held-out variants of those scenarios without feedback or reflection. We repeat this sequence with three seeds, using the same task order for both arms and all seeds.

Exo reduces held-out model cost by 61% at the same observed pass rate, with savings that also cover the learning period. Both arms pass 96.9% of held-out tasks. Exo costs $0.141 per task versus $0.359, uses 27% fewer model rounds, and reduces mean task time by 13% (Fig. 6). Including learning and reflection, mean total cost falls from $57.36 to $32.70, a 43% reduction.

Inspection of the learned harness shows reusable API-documentation tools and changes to context construction.

Table 1. Graded-task outcomes for Exo vs identical policy without policy editing (Control). Arrows indicate the preferred direction. AppWorld and CLB values are means; SREGym reports median time and rounds and mean cost. CLB costs include reflection.

Metric	Control	Exo
AppWorld Held-out tasks; 3 seeds		
Tasks passed ↑	**96.9%**	**96.9%**
Model cost / task ↓	$0.359	**$0.141**
Time / task ↓	36.5 s	**31.9 s**
SREGym 20 held-out incidents		
Diagnosis time ↓	81 s	**53 s**
Model rounds / incident ↓	16	**10**
Model cost / incident ↓	$0.892	**$0.584**
CLB coding All tasks; 5 runs		
Issues resolved ↑	**100%**	**100%**
Shell steps / issue ↓	7.42	**6.28**
Model cost / issue ↓	$0.507	**$0.436**

For example, `appworld_api_docs` fetches compact documentation for several relevant APIs in one call, avoiding repeated discovery. Exo also moves its changing task list out of the standing prompt and revises context construction; cached input share rises from 57.5% to 86.1%. Fig. 7 traces seed 2: Exo revises its harness after each of the first three tasks, and its cumulative cost falls below Control's at task 9. The learned procedures differ across seeds. Seeds 1 and 2 build broadly reusable tools, while seed 3 creates many

scenario-specific tools and spends more during learning. All three nevertheless achieve lower held-out cost. This variation shows that specialization need not converge to a single implementation.

5.4.2 SREGym

SREGym (Clark et al., 2026) evaluates agents diagnosing and mitigating faults injected into running services. The efficiency comparison uses 21 learning incidents with feedback and reflection, followed by 20 unseen incidents from the same fault families on different applications, without feedback or reflection.

Exo's learned policy reduces diagnosis time, model rounds, and the cost of handling subsequent incidents. Median diagnosis time falls from 81 to 53 seconds (35%), median model rounds from 16 to 10 (38%), and mean model cost per held-out incident from $0.892 to $0.584 (35%; Tab. 1). Correctness is comparable across additional held-out runs.

Exo builds and repeatedly refines a diagnostic tool that packages evidence needed across incidents. Exo creates and refines an incident-snapshot tool that collects unhealthy pods, revision and template anomalies, and suspicious log lines, revising this tool five times and reusing it as its first diagnostic step on held-out incidents. The tool consolidates evidence that otherwise requires multiple shell calls, giving subsequent diagnoses a reusable starting procedure. It builds an additional 5 tools for targeted investigation.

5.4.3 Continual Learning Bench Coding

On CLB coding tasks, Exo uses 15% fewer shell steps and 14% less model cost, with both policies resolving every issue. We evaluate a fixed sequence of 19 issues from `tablib` and `tenacity` using Continual Learning Bench (Asawa et al., 2026), with five runs per arm. Both arms reflect after each graded issue; there is no separate held-out phase. Exo averages 6.28 shell steps and $0.436 per issue, versus 7.42 and $0.507 for the control, including reflection. Its observed self-changes are tool and skill edits between issues.

Limits of adaptation. Access to self-editing provides a benefit only when the model is capable enough to discover and reuse effective changes. In additional experimentation running Exo on AppWorld with `gpt-5-mini`, three seeds made limited harness edits and did not reuse their created tools on held-out tasks. Total cost and pass rates were similar ($6.63 and 59% vs $6.73 and 55% for Exo vs control). Thus, we believe there is an intelligence floor necessary to reap the benefits of Exo's architecture.

6 Related Work

Learning from execution feedback. Execution feedback can guide future behavior through retained lessons (Shinn et al., 2023; Zhao et al., 2024; Ouyang et al., 2025) or revisions to prompts and workflows (Zhang et al., 2025b; Agrawal et al., 2025; Yuksekgonul et al., 2024; Cheng et al., 2024). The Self-Defining Operator focuses on incident response, learning reusable playbooks and detectors (Li et al., 2026). In Exo, the running agent can also revise the code governing when and how it learns from feedback.

Harness construction and self-modification. Harness search automates policy generation and refinement (Lee et al., 2026; Lou et al., 2026; Xia et al., 2026), including evolution over unlabeled test batches (Nie et al., 2026). JIT-Agent generates task-conditioned harnesses with immutable history and mutable controller state (Zhang et al., 2026a). Self-modification spans changes to running code (Yin et al., 2024), benchmark-driven agent evolution (Zhang et al., 2025a), and editable improvement procedures (Zhang et al., 2026b). Concurrent work, Harness^2, refines task harnesses without ground-truth verifiers and updates an improvement harness across tasks (Xu et al., 2026). Exo couples self-modification to task continuation: an accepted revision continues the task, retaining its history and environment.

Within-task adaptation differs in what remains fixed. Live-SWE-agent creates tools while retaining its underlying agent loop (Xia et al., 2025); Continual Harness periodically invokes a refiner over prompts, sub-agents, skills, and memory (Karten et al., 2026). Exo also exposes the executor code governing context construction, model calls, and refinement. The acting agent can change both these behaviors and the policy deciding when and how to revise them.

Runtime support for adaptation. Runtime adaptation requires both extensibility and independent control over state and execution. DeepSeek Harness combines replaceable plugins with an append-only event log (DeepSeek-AI, 2026), using Cordis for hot replacement and reversible component effects (Shi et al., 2026). Independent checks govern which actions and changes take effect: LogAct couples a shared log with decoupled voters admitting task actions (Balakrishnan et al., 2026), while Self-Healing Harness admits behavioral rules through replay or forward trials (Tayebati et al., 2026). Exo uses this separation to let the agent replace the policy governing an ongoing task while keeping execution history and replacement authority outside that policy's control.

7 Discussion

Building bitter-lesson-aligned systems. As reasoning capabilities grow, we believe it is important that the systems we build around them provide benefits that persist and scale proportionally with that growth, rather than being obviated

as per the Bitter Lesson (Sutton, 2019). Doing so requires a reprioritization of traditional design goals: *flexibility*, *modifiability*, and *legibility* must be elevated to first-class design principles. In the limit, this suggests placing all harness policy, including how it improves itself, under model control, retaining only the mechanisms needed for safe self-evolution. Exo pursues this design with a minimal protected core and an editable, inspectable policy layer. Notably, this does not prescribe a "thin" harness: the agent can and does build substantial domain-specific tooling or remove machinery it no longer needs. The appropriate harness can change with both the model and its workload.

Runtime legibility enables better decision making. In order to make self-improving systems effective, they need visibility into their own behavior. Policy source code exposes what can be changed, while queryable history records what happened and cost, latency, and cache annotations expose resources consumed. These observations provide denser reward signals about execution efficiency, helping the agent assess and guide improvements before task completion. We believe this to be a key reason for Exo's effectiveness. Preserving history across rollbacks likewise keeps evidence of failed attempts available to future decisions.

A new timescale of system development. Historically, the time to build a system has been longer than the timescale of an individual task. As model-assisted development has become faster and task horizons grow, this assumption has begun to flip, and constructing useful specialization can now take less time than the single task that motivates it. Thus, we must increasingly consider development as something that can continue during execution, with procedures synthesized just in time to address the workload at hand. Exo is motivated by this trend, and we see this occur in its evaluations on long-running open-world tasks (§5.2), with investment in tooling and policy edits paying back before the objective is complete (§5.2.1). Learning in the deployment environment also grounds revisions in the conditions where they will be used, reducing dependence on representative benchmarks and faithful emulation. This suggests a broader role for continually adapting systems, with trust in their revisions becoming a central systems challenge.

Generalizing guardians. Exo's guardian separates the authority to propose a revision from the authority to activate it. The minimal check we provide as default in our implementation is build success; richer guardians could additionally do closer LLM-as-a-judge policy inspection, or even trials in forked environments. The agent could propose additional tests as failures emerge, while changes to the governing acceptance criteria remain independently authorized. More extensive validation incurs additional cost. An open question is how to match validation effort to a revision's consequences while retaining an independent enforcement boundary. We leave exploring additional guardian mechanisms to future work.

Interpretable continual learning. Exo's adaptations appear as explicit changes to code, prompts, and skills. Compared with continual learning encoded solely in model weights, these artifacts expose the learned procedures directly, allowing agent operators to inspect, test, and revise them. Changes can also be examined alongside the execution history that motivated them. This is particularly valuable in domains where operators need to understand, audit, and authorize changes to system behavior. Future guardians could present proposed revisions, supporting evidence, and validation results to a human for approval or rejection before activation. This would let organizations selectively adopt learned procedures through existing review processes.

Exo as an agent runtime. Exo's agent runtime decomposition into executor and protected state provides additional benefits in a practical deployment setting. Agents can switch between executor implementations while retaining their durable history and artifacts. Shared runtime services support multiple agents executing concurrently on a single node, each with its own policy and state. Sandbox execution can be offloaded to remote providers, while interchangeable environment backends let deployments change where and how task actions execute without rewriting agent policy. These capabilities allow executor hosting, persistence, and environment resources to be provisioned separately. We leave reporting on a systematic evaluation of agent density, throughput, and resource efficiency to future work.

Future work. A natural next step from Exo's current architecture is exploring parallel self-improvement through explicit agent lineages: fork canonical state, record the parent–child relationship in the event log, and provide a communication channel between branches. Agents could then explore revisions concurrently and share findings with traceable provenance. We leave this to future work.

8 Conclusion

We presented Exo, an agent runtime architecture for in-harness learning and adaptation during ongoing work. Exo collapses the optimization loop into task execution, allowing agents to revise all aspects of their harness at runtime. Protected mechanisms preserve execution history and govern executor replacement, while learned adaptations remain explicit and inspectable. Across open-world objectives and graded benchmarks, our evaluation demonstrates efficiency gains during ongoing work and reuse of learned procedures on subsequent tasks. Exo offers a path toward systems whose policies can evolve with their workloads and the capabilities of their underlying models.

REFERENCES

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. GEPA: Reflective prompt evolution can outperform reinforcement learning. arXiv:2507.19457, 2025. URL https://arxiv.org/abs/2507.19457.

Anthropic. Effective context engineering for AI agents, September 2025. URL https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents.

Anthropic. Claude Code, 2026. URL https://claude.com/product/claude-code. Web resource consulted September 9, 2026.

Asawa, P., Glaze, C. M., Orlanski, G., Ramakrishnan, R., Xu, B., Biswal, A., Chen, V. S., Sala, F., Zaharia, M., and Gonzalez, J. E. Continual Learning Bench: Evaluating frontier AI systems in real-world stateful environments. arXiv:2606.05661, 2026. URL https://arxiv.org/abs/2606.05661.

Astrom, K. J. and Murray, R. M. *Feedback systems: An introduction for scientists and engineers*. Princeton University Press, Princeton, NJ, April 2008.

Balakrishnan, M., Bharambe, A., Testuggine, D., Geraghty, D., Mao, D., Venkat, V., Mironov, I., Baradi, R., Aiyer, G., and Dudin, V. LogAct: Enabling agentic reliability via shared logs. arXiv:2604.07988, 2026. URL https://arxiv.org/abs/2604.07988.

Bittker, M. RS-SDK: RuneScape automation library for coding agents. Software repository, 2026. URL https://github.com/MaxBittker/rs-sdk. Accessed October 29, 2026.

Carlini, N. Building a C compiler with a team of parallel Claudes. Anthropic Engineering, February 2026. URL https://www.anthropic.com/engineering/building-c-compiler.

Cheng, C.-A., Nie, A., and Swaminathan, A. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. arXiv:2406.16218, 2024. URL https://arxiv.org/abs/2406.16218.

Clark, J., Su, Y., Pial, S. M. R., Tian, Y., Gniedziejko, L., Jacobsen, H.-A., Chen, Y., and Xu, T. SREGym: A live benchmark for AI SRE agents with high-fidelity failure scenarios. arXiv:2605.07161, 2026. URL https://arxiv.org/abs/2605.07161.

DeepSeek-AI. DeepSeek Harness: Everything is a plugin. Software and architecture documentation, 2026. URL https://github.com/deepseek-ai/deepseek-harness. Accessed October 27, 2026; source revision 22ddee3792402c3e70ec5f86a63c3e3e895a2f60.

Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 251–266, New York, NY, USA, 1995. Association for Computing Machinery. doi: 10.1145/224056.224076.

Exo Contributors. Exo. Software repository, commit c5c1963, 2026. URL https://github.com/exoharness/exo/tree/c5c1963a3445417e64938c139da88edd9154075d.

Jagex. RuneScape, 2026. URL https://www.runescape.com/. Web resource consulted October 29, 2026.

Karten, S., Zhang, J., Upaa, T., Feng, R., Li, W., Shi, C., Jin, C., and Vodrahalli, K. Continual harness: Online adaptation for self-improving foundation agents. arXiv:2605.09998, 2026. URL https://arxiv.org/abs/2605.09998.

Krentsel, A., Agarwal, S., Cemri, M., Liu, S., Sankhe, S., Mao, Z., Zaharia, M., and Stoica, I. Reality is the final verifier: On two key gaps in agentic software engineering, 2026. URL https://arxiv.org/abs/2609.12039.

Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., and Finn, C. Meta-Harness: End-to-end optimization of model harnesses. arXiv:2603.28052, 2026. URL https://arxiv.org/abs/2603.28052.

Li, S., Anderson, T., Mahajan, R., Peter, S., and Zettlemoyer, L. A self-defining operator for distributed applications. arXiv:2610.14547, 2026. URL https://arxiv.org/abs/2610.14547.

Lin, W. Towards self-driving codebases. Cursor Research, February 2026. URL https://cursor.com/blog/self-driving-codebases.

Liu, S., Cemri, M., Agarwal, S., Krentsel, A., Naren, A., Mang, Q., Li, Z., Gupta, A., Maheswaran, M., Cheng, A., Pan, M., Boneh, E., Ramchandran, K., Sen, K., Zaharia, M., Dimakis, A. G., and Stoica, I. SkyDiscover: A flexible, adaptive framework for AI-driven scientific and algorithmic discovery. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pp. 1223–1227. Association for Computing Machinery, 2026. doi: 10.1145/3786335.3813221. URL https://doi.org/10.1145/3786335.3813221.

Lou, X., Lázaro-Gredilla, M., Dedieu, A., Wendelken, C., Lehrach, W., and Murphy, K. P. AutoHarness: Improving LLM agents by automatically synthesizing a code harness. arXiv:2603.03329, 2026. URL https://arxiv.org/abs/2603.03329.

METR. Task-completion time horizons of frontier AI models, 2026. URL https://metr.org/time-horizons/.

Nie, J., Zhang, Y., Song, J., Cai, Q., Yu, D., Guo, Y., Tian, X., and Han, B. TTHE: Test-time harness evolution. arXiv:2607.08124, 2026. URL https://arxiv.org/abs/2607.08124.

Nous Research. Hermes Agent. Software repository, 2026. URL https://github.com/NousResearch/hermes-agent. Web resource consulted September 9, 2026.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. Introducing Codex, May 2025a. URL https://openai.com/index/introducing-codex/.

OpenAI. Building more with GPT-5.1-Codex-Max, November 2025b. URL https://openai.com/index/gpt-5-1-codex-max/.

OpenAI. Codex CLI. Software repository, 2026a. URL https://github.com/openai/codex. Accessed October 29, 2026.

OpenAI. On the Navier–Stokes millennium prize problem, September 2026b. URL https://openai.com/index/navier-stokes-solution/.

OpenClaw. OpenClaw, 2026. URL https://openclaw.ai/. Web resource consulted September 9, 2026.

Ouyang, S., Yan, J., Hsu, I.-H., Chen, Y., Jiang, K., Wang, Z., Han, R., Le, L. T., Daruki, S., Tang, X., Tirumalashetty, V., Lee, G., Rofouei, M., Lin, H., Han, J., Lee, C.-Y., and Pfister, T. ReasoningBank: Scaling agent self-evolving with reasoning memory. arXiv:2509.25140, 2025. URL https://arxiv.org/abs/2509.25140.

Pace. The AI operations platform for insurance, 2026. URL https://withpace.com/. Web resource consulted September 9, 2026.

Pace Team. Introducing Tempo, July 2026. URL https://withpace.com/news/tempo.

Pan, M. Z., Arabzadeh, N., Cogo, R., Zhu, Y., Xiong, A., Agrawal, L. A., Mao, H., Shen, E., Pallerla, S., Patel, L., Liu, S., Shi, T., Liu, X., Davis, J. Q., Lacavalla, E., Basile, A., Yang, S., Castro, P., Kang, D., Sen, K., Song, D., Gonzalez, J. E., Stoica, I., Zaharia, M., and Ellis, M. Measuring agents in production. arXiv:2512.04123, 2025. URL https://arxiv.org/abs/2512.04123v4. Version 4, revised June 4, 2026.

Resolve.ai. AI for prod, 2026. URL https://resolve.ai/. Web resource consulted September 9, 2026.

Shi, Y., Zhang, W., and Cui, T. A programming paradigm for spatiotemporal composability. arXiv:2608.25512, 2026. URL https://arxiv.org/abs/2608.25512.

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. arXiv:2303.11366, 2023. URL https://arxiv.org/abs/2303.11366.

Sierra. How SiriusXM drives listener loyalty with Sierra, 2026. URL https://sierra.ai/customers/siriusxm. Web resource consulted September 9, 2026.

Sutton, R. S. The bitter lesson, March 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Tayebati, S., Kumar, D., Darabi, N., Krishnan, R., and Trivedi, A. R. Self-healing harness for runtime oversight of agent self-modification. arXiv:2609.24130, 2026. URL https://arxiv.org/abs/2609.24130.

Trivedi, H., Khot, T., Hartmann, M., Manku, R., Dong, V., Li, E., Gupta, S., Sabharwal, A., and Balasubramanian, N. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Xia, C. S., Wang, Z., Yang, Y., Wei, Y., and Zhang, L. Live-SWE-agent: Can software engineering agents self-evolve on the fly? arXiv:2511.13646, 2025. URL https://arxiv.org/abs/2511.13646.

Xia, P., Han, R., Wang, Z., Chen, Y., Zhuang, Y., Lee, Y., Huang, C., Yu, H., CuiZhu, Z., Ming, Y., Yao, H., Gokturk, B., Pfister, T., and Lee, C.-Y. RRSI: Regularized recursive self-improvement of agent harnesses. arXiv:2609.24972v3, 2026. URL https://arxiv.org/abs/2609.24972v3.

Xu, R., Chen, Y., CuiZhu, Z., Mishra, B. D., Ming, Y., Yu, H., Han, R., Lee, C.-Y., and Pfister, T. Harness^2: Recursive agent harnessing for an open world. *arXiv preprint arXiv:2610.14224*, 2026. URL https://arxiv.org/abs/2610.14224.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Yin, X., Wang, X., Pan, L., Lin, L., Wan, X., and Wang, W. Y. Gödel agent: A self-referential agent framework for recursive self-improvement. arXiv:2410.04444, 2024. URL https://arxiv.org/abs/2410.04444.

Yuksekgonul, M., Bianchi, F., Boen, J., Liu, S., Huang, Z., Guestrin, C., and Zou, J. TextGrad: Automatic "differentiation" via text. arXiv:2406.07496, 2024. URL https://arxiv.org/abs/2406.07496.

Zechner, M. Pi: Coding agent. Software repository, 2026. URL https://github.com/earendil-works/pi. Accessed October 29, 2026.

Zhang, G., Lu, L., Xie, F., Zhu, K., Wang, J., Xie, Z., Yu, Z., Liu, Z., Sun, Z., Li, Q., Liao, Y., Chang, H., Hu, X., Ren, Q., Zhou, W., Hu, C., Deng, Y., and Yan, S. JIT-Agent: Scaling harness intelligence via just-in-time harness evolution. arXiv:2608.25593v2, 2026a. URL https://arxiv.org/abs/2608.25593v2.

Zhang, J., Hu, S., Lu, C., Lange, R., and Clune, J. Darwin gödel machine: Open-ended evolution of self-improving agents. arXiv:2505.22954, 2025a. URL https://arxiv.org/abs/2505.22954.

Zhang, J., Zhao, B., Yang, W., Foerster, J., Clune, J., Jiang, M., Devlin, S., and Shavrina, T. Hyperagents. arXiv:2603.19461, 2026b. URL https://arxiv.org/abs/2603.19461.

Zhang, Q., Hu, C., Upasani, S., Ma, B., Hong, F., Kamanuru, V., Rainton, J., Wu, C., Ji, M., Li, H., Thakker, U., Zou, J., and Olukotun, K. Agentic context engineering: Evolving contexts for self-improving language models. arXiv:2510.04618, 2025b. URL https://arxiv.org/abs/2510.04618.

Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y.-J., and Huang, G. ExpeL: LLM agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38 (17):19632–19642, 2024. doi: 10.1609/aaai.v38i17.29936. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

AI USAGE DISCLOSURE

The authors used Claude Code and Codex to assist with inspecting and organizing experiment results and some figure rendering. All coding was done collaboratively with Claude Code or Codex with models varying from April 2026 through July 2026, with human-written design documents and collaborative programming. Paper writing was done primarily by the authors, using codex to assist in revising passages for clarity or brevity, and formatting.

A Preservation Proof and Scope

We use the main paper's state $s_t = (d_t, e_t, \pi_t)$, protected transition mechanism K, and guardian G.

Preservation under self-editing. If the initial state satisfies the storage contract, every operation affecting protected state is mediated by K, and every permitted transition preserves the contract, then it holds after any finite sequence of operations and executor replacements.

The proof is induction on transitions. The initial state satisfies the contract by assumption. If it holds at step t, applying K preserves append-only history and prior artifact versions at step $t + 1$, including when the operation replaces π_t. Induction establishes preservation for any finite execution, independently of how the executor chooses edits.

Similarly, if $G(s, \pi') = 1$ implies a candidate property $C(\pi')$ and the initial executor satisfies C, every installed executor satisfies C at installation: a transition either retains the current implementation or installs one accepted by G.

Enforcement assumptions and limits. The storage and replacement mechanisms must remain outside the executor's editing authority. Direct storage access or unrestricted administrative deletion would invalidate complete mediation; the broadly writable development configuration does not enforce these restrictions. Guardian acceptance establishes only the properties actually checked: build success alone guarantees neither task correctness nor improvement. Recovery from a hung executor requires independent authority to stop it and activate a fallback; preservation alone does not ensure liveness. Scoped storage keeps credentials outside sandbox files, but model-context exposure depends on authorized reads and executor tools. We provide such authorized reads, but do not describe their implementation here as it is out of scope for this paper.

Recovery scope. Policy and sandbox rollback preserve canonical history and record the rollback. Snapshot scope depends on the provider: Docker image snapshots omit process memory and host-mounted source. Neither reverses external effects, which require domain-specific recovery.

B Open-World Measurement Details

Tab. A1 reports one standalone/transfer pair per objective and harness. Completion costs use the first recorded endpoint at or above the target. Money-making uses the last sampled turn before eight active hours, excluding logged outages.

Table A1. Open-world outcomes, one run per entry. Combat/crafting report completion cost; money-making is sampled before eight active hours. Shading compares harnesses within each row and run group: green (best), orange (intermediate), red (worst).

	Standalone			**Transfer**		
Metric	Exo	Codex	Pi	Exo	Codex	Pi
Combat ($) ↓	**3.71**	7.26	7.95	**1.92**	6.87	10.89
Crafting ($) ↓	**2.99**	3.90	3.41	**0.88**	1.89	1.61
Money ($) ↓	**2.84**	5.14	5.08	**2.84**	3.68	4.08
Coins/$ (k) ↑	**332**	231	148	**334**	265	210