PERDURA: STATEFUL CODE EXECUTION HARNESS FOR LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon agents must generate code, invoke tools, and incorporate feedback across extended tasks. Many existing harnesses leave part of this execution continuity in the model-visible interaction history, requiring the model to recover prior outcomes, available computational state, and unfinished work as trajectories grow. We propose **Perdura**, a stateful execution harness in which the model chooses executable Python actions, while the runtime owns execution continuity. Across steps, the runtime retains intermediate computational state, exact records of prior operations, and unresolved work through a persistent namespace, an execution ledger, and an explicit work lifecycle. Each model call receives only a bounded context derived from runtime state and the current observation. Across three long-horizon benchmarks and four widely used agent harnesses under the same base model (GLM-5.3 Flash), Perdura attains task success comparable to the top-scoring harness on each benchmark while using 26–53% fewer input tokens than that harness, and no harness achieves both a higher score and a lower cost on any benchmark. Trajectory analyses show how computational reuse, registered waits, and recorded execution evidence support continuation across steps.

1 INTRODUCTION

Long-horizon agents rarely start each step from a clean slate. As agent workflows stretch across more steps, tools, and waiting periods, later actions increasingly depend on execution state created earlier in the trajectory (Dong et al., 2026; Meng et al., 2026). A program written early in a task may produce objects that later steps reuse; a tool call may create an external effect whose outcome matters hours later; a pending job or approval may remain unresolved while other work proceeds. As these dependencies accumulate, generating the next action becomes only one part of continued execution. The agent must also preserve or recover what has already happened, what computational state remains available, and what work is still unfinished.

Many agent harnesses leave part of this continuity inside the model-visible interaction history. Intermediate values may need to be recomputed or serialized manually; prior tool outcomes must be recovered from earlier messages or summaries; asynchronous work is often tracked through polling, shell processes, or textual reminders. As histories grow, truncation and summarization can further separate the model from exact execution evidence (Qiao et al., 2023; Chen et al., 2026; May et al., 2026; Packer et al., 2023; Wang et al., 2026b; Lin et al., 2026). These mechanisms can work, but the model often has to reconstruct execution state that earlier steps had already established (Figure 1).

This suggests a different division of responsibility. The model should decide what executable action to take next, such as running code, calling a tool, sending a message, or waiting for an event; the runtime should preserve the state those actions create, including variable bindings, operation outcomes, and unfinished work. The runtime is not a second agent: the model chooses what to do; the runtime validates and commits transitions, schedules wakeups, and restores state after interruption.

We introduce **Perdura**, a stateful execution harness that makes this model–runtime division of responsibility explicit through two coupled layers: a model-facing execution layer and a stateful runtime. The execution layer gives the model a unified Python interface for executable actions over a persistent namespace of values, functions, and runtime-provided handles. Through this interface, the model can compute, invoke tools, communicate, manage unfinished work, and wait for external events. The runtime validates and executes these actions while preserving the state they create across

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.

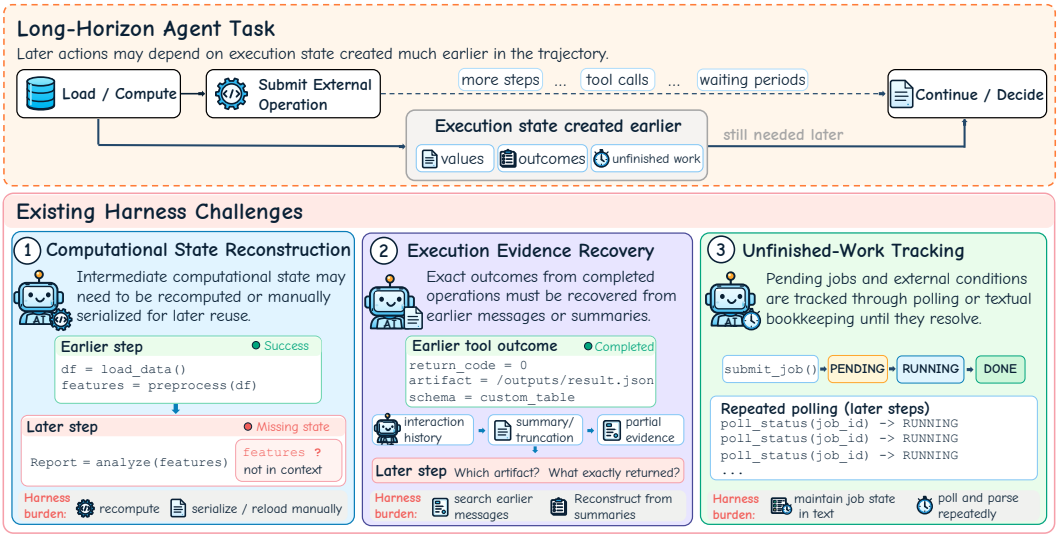


Figure 1: **Long-horizon tasks create cross-step dependencies on execution state.** Existing harnesses may require the agent to reconstruct intermediate computational state, recover exact execution evidence from interaction history or summaries, and track unfinished work through textual bookkeeping and repeated polling. These execution-management burdens waste computation and context and increase the risk of downstream errors.

steps: it maintains the namespace, records tool effects as durable execution evidence, and manages unfinished work through explicit lifecycles. At each step, the runtime constructs a bounded context from committed runtime state and the current observation, so the model can continue without reconstructing the full interaction history. When a registered event ends a wait, the runtime updates task state and constructs the next bounded context. If the Python process is replaced, it restores recoverable namespace state without replaying historical code (Figure 2).

The runtime is where Perdura's central architectural novelty lies: it takes explicit responsibility for the execution state that later actions depend on. We organize this state around three execution-continuity domains. The *persistent namespace* (N) maintains computational state available for continued execution, including values, functions, and runtime-bound handles to external resources; the runtime preserves this state across steps while exposing only a compact description of it in the bounded context. The *execution ledger* (L) preserves durable evidence of prior execution, including typed messages, operation requests and outcomes, evidence of external effects, and code history outside the context window, so earlier evidence can be retrieved directly rather than reconstructed from summaries or replayed computation. The *work lifecycle* (W) represents unfinished commitments, including WorkItems, dependencies, completion conditions, and registered waits, allowing execution to suspend without model polling and resume when the relevant event or condition arrives.

Perdura builds on code-based action interfaces (Wang et al., 2024), externalized memory (Packer et al., 2023), and coding agents such as Claude Code and Codex that support extended, multi-step tool use (Anthropic, 2026; OpenAI, 2026). Prior systems persist separate individual forms of agent state: interpreter state (Qiao et al., 2023; Ran et al., 2026), an event log with a persistent namespace (Lin et al., 2026), a session with pending work (Wen et al., 2026), or recovery from ambiguous outcomes (Zhang, 2026). Perdura combines these forms of state in one runtime but keeps them separate, reflecting their different commit, recovery, and failure semantics.

We evaluate Perdura across three long-horizon benchmarks (SENTINELBENCH, TOOLMAZE, and BEAM), four comparative harnesses, and a shared base model (GLM-5.3 Flash) (Maldaner et al., 2026; Zhu et al., 2026; Tavakoli et al., 2025). We measure task completion and model-token cost, and examine recorded trajectories for computational reuse, evidence access, and unfinished-work continuation. Perdura attains task success comparable to the top-scoring harness on each benchmark at 47–74% of its cost, and lies on every benchmark's score–cost Pareto frontier. Our contributions are threefold:

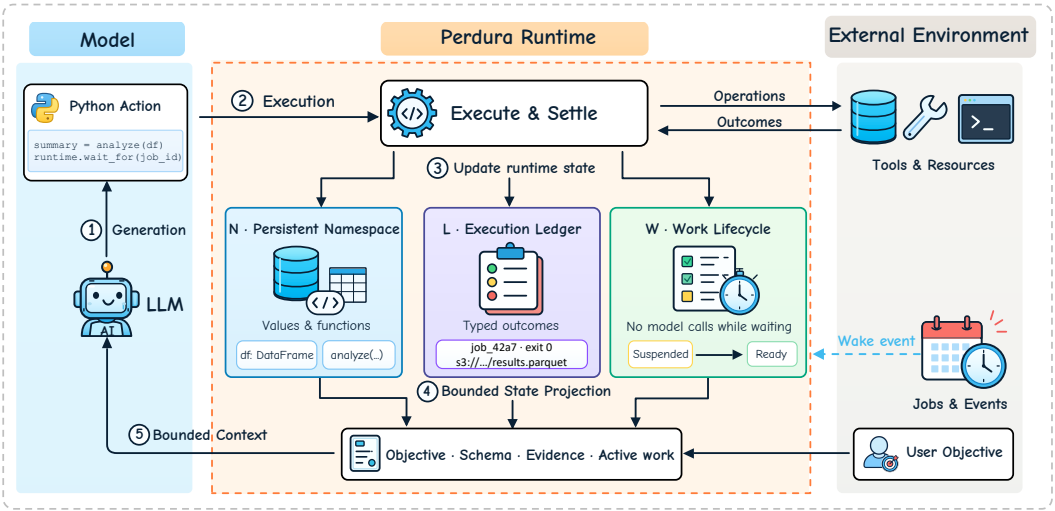


Figure 2: **Overview of the Perdura stateful execution harness.** The model proposes executable Python actions from a bounded context c_t. The runtime manages execution and settlement across three domains: computational continuity in the persistent namespace (N), durable execution evidence in the execution ledger (L), and unfinished work in the work lifecycle (W). The context projection Π constructs a bounded context from runtime state and the current observation, presenting concise state and evidence without exposing the full interaction history.

- **Execution-state separation.** We identify three classes of cross-step execution state, computational state, execution evidence, and unfinished work, that obey different commit, recovery, and failure semantics, and should therefore be kept distinct by the runtime.
- **Perdura.** We build a harness that couples a Python action interface with a runtime that maintains a persistent namespace (N), an execution ledger (L), and a work lifecycle (W), enforces their boundaries through namespace generations, Operation settlement, and the WorkItem lifecycle, and gives the model a bounded context.
- **End-to-end and trajectory evaluation.** Across three benchmarks, four comparative harnesses, and GLM-5.3 Flash, we report task outcomes and efficiency, and examine representative trajectories for state reuse, evidence access, and work continuation.

2 Problem Setting and Formal Requirements

2.1 Long-horizon task formulation

An agent pursues a user objective u_0 by acting on a partially observable environment

$$\mathcal{M}_{\text{env}} = (\mathcal{S}_{\text{env}}, \mathcal{A}_{\text{env}}, \mathcal{O}, \mathcal{P}_{\text{env}}, \mathcal{Z}_{\text{env}}, \mathcal{R}), \tag{1}$$

with external states, agent actions, observations, transition and observation kernels, and task feedback; an action may trigger several lower-level operations, and the agent observes outcomes rather than the external state. Execution produces a trajectory

$$\tau = (u_0, o_0, a_0, o_1, a_1, \ldots, a_{T-1}, o_T), \tag{2}$$

where $a_t \in \mathcal{A}_{\text{env}}$ is an agent action, o_{t+1} the observation that follows it, and T the number of steps. At step t, a harness gives the model a context $c_t = f(h_t)$ built from the history $h_t = (u_0, o_0, a_0, \ldots, o_t)$, and the model chooses $a_t \sim \pi_\theta(\cdot \mid c_t)$. A task is *long-horizon* when a_t depends on information produced many steps earlier (Dong et al., 2026; Meng et al., 2026). Since h_t grows with t while c_t must fit a bounded context, the harness must decide what reaches c_t: parts of h_t and state that earlier actions created outside it. We call this information the *execution state*.

2.2 THE EXECUTION-MANAGEMENT BURDEN

Execution state has three components, by origin. **Computational state** is produced by the agent's own code and lives in the interpreter rather than in h_t; live resources may not be serializable. **Execution evidence** consists of prior operation outcomes; it appears in h_t but must remain exact as truncation or compaction shortens the history (Packer et al., 2023; Wang et al., 2026b; Lin et al., 2026). **Unfinished work** is a commitment whose completion depends on a future event, and nothing in h_t records when to resume. With the transcript as the only carrier, all three are rebuilt from text (Figure 1): computation is re-run, evidence is re-read or lost, and pending work is re-polled.

2.3 FORMAL REQUIREMENTS

Removing this burden requires the runtime, rather than the transcript, to carry execution state. R1 follows from the bounded context above, and R2–R4 require each of its three components to be carried outside c_t. Carrying state outside the transcript adds an obligation: the runtime's state must stay consistent with an external world it cannot roll back, including across failures, which R5 and R6 state. **(R1) Bounded and inspectable model context**: the model-visible context remains bounded while the runtime records what was exposed at each invocation; **(R2) Computational-state continuity**: state needed by later actions remains recoverable across interactions and runtime replacement; **(R3) Durable execution evidence**: exact execution evidence remains available outside the bounded model context; **(R4) Explicit unfinished work**: pending commitments and wait conditions are represented explicitly, enabling suspension without model polling; **(R5) Explicit effect settlement**: an operation's execution is distinguished from its established outcome, and uncertain effects are surfaced to the agent rather than retried automatically; and **(R6) Recoverable execution**: execution resumes from recoverable state without replaying historical code.

2.4 RESEARCH QUESTIONS

The requirements imply two claims: runtime-held execution state should keep task performance at lower cost (RQ1), and each component should be carried as required (RQ2: R1–R2; RQ3: R3–R4).

RQ1 (Task outcomes and cost): What task completion and model-token cost does Perdura achieve?

RQ2 (Computational-state continuity, R1–R2): How does Perdura keep computational state created in one step available to later steps while the model sees only a bounded context?

RQ3 (Execution continuity, R3–R4): How does Perdura continue execution when later actions depend on execution evidence or unfinished work from earlier steps?

The benchmarks do not exercise R5 and R6, so we treat them as design properties (§3.3, §3.5).

3 PERDURA: STATEFUL CODE EXECUTION

Perdura instantiates the requirements in Section 2 with a stateful execution runtime. Section 3.1 defines the runtime state and the step loop; Sections 3.2–3.5 describe its four components.

3.1 RUNTIME STATE AND STEP LOOP

Perdura's runtime state, distinct from the environment of Equation 1, is

$$\bar{r}_t = (N_t, L_t, W_t, Q_t). \tag{3}$$

N_t, L_t, and W_t are the three execution-continuity domains: the persistent namespace maintains computational state, the execution ledger preserves execution evidence, and the work lifecycle represents unfinished work. Q_t is the runtime's control state, which hosts resources, schedules waits, and recovers the other three. These are roles within one runtime state, not separate modules. R1 is realized by the context projection below, R2 by the namespace (§3.2), R3 and R5 by the ledger (§3.3), R4 by the work lifecycle (§3.4), and R6 by the control state (§3.5).

Each model invocation and its selected code form a runtime-managed step (Figure 2). Before invoking the model, Perdura constructs a bounded context from its current runtime state and observation:

$$c_t = \Pi(\bar{r}_t, o_t) = \Big[\Pi_N(N_t), \Pi_L(L_t), \Pi_W(W_t), \Pi_Q(Q_t), \Pi_O(o_t)\Big]. \tag{4}$$

Π decides what the model sees without placing the full runtime state in the prompt; Perdura thus instantiates the context function f of Section 2 over runtime state rather than over the history h_t. It replays a retained window of recent steps and summarizes the rest: Π_N lists variables by name, type, and a bounded excerpt, older code appears only as code history, and earlier evidence is retrieved by the model on demand through `task.search` rather than injected. Π_W shows active WorkItems and their waits, Π_Q the task definition and callable surface, and Π_O the current observation and new messages. The inputs of each step's context are committed to the ledger before the model is invoked, so the model-visible input can be reconstructed later. The model then produces

$$y_t = (\mathrm{intent}_t, \hat{a}_t) \sim \pi_\theta(\cdot \mid c_t), \tag{5}$$

where intent_t is a short natural-language statement of what the step will do and $\hat{a}_t$ is a proposed executable program, not yet an established external action.

Perdura validates the proposed program $\hat{a}_t$ and executes it if admitted; during execution, the code may make several calls through typed syscall boundaries, summarized by the transition

$$(x_{t+1}, \bar{r}_{t+1}, o_{t+1}) \sim \mathcal{T}_{\mathrm{commit}}(x_t, \bar{r}_t, \hat{a}_t, \mathcal{E}_{\mathrm{ext}}). \tag{6}$$

Here $x_t \in \mathcal{S}_{\mathrm{env}}$ is the external state, $\bar{r}_t$ is Perdura's runtime state, and $\mathcal{E}_{\mathrm{ext}}$ denotes external events and operation outcomes. The transition includes validation, execution, and settlement. A rejected proposal can leave external state unchanged while still producing runtime evidence.

3.2 Persistent namespace (N)

Computational state produced by the agent's code lives in the interpreter, not in the transcript; a transcript-based harness must either re-run the code or serialize its results into text. Perdura keeps this state in a persistent namespace $N_t = (\mathcal{V}_{\mathrm{value}}, \mathcal{F}, \mathcal{H}_{\mathrm{binding}})$ of values, functions, and Handles. After each step, recoverable variables and functions are saved as a namespace version that a later process can restore, and non-recoverable values are unavailable after replacement. The namespace is not rolled back: a step that fails partway keeps the updates it made, because the effects of that code have already occurred and a rollback would open a gap between the namespace and the world. The namespace projection Π_N exposes only a compact description, e.g., names, types, shapes, signatures, and short excerpts, while full values remain available to later code.

Some state cannot be a value: a browser page, an MCP connection, a shell session, or the result of a call that has not yet returned. Perdura represents such state as a *Handle*, which plays two roles. It is a reference to a live resource: the resource itself is held in the registry of Q_t, while N_t holds only the Handle, so its identity survives steps and process replacement. It is also an asynchronous callback channel: a long-running call returns a Handle immediately, and when its result or a registered condition arrives, the runtime delivers a `handle.ready` message that can wake the task (§3.4), so a step need not block on an external event.

3.3 Execution ledger (L)

To act on a changing world, the agent must know precisely what its earlier actions did. In a transcript, a proposed call, its execution, and its effect are blurred together: a timeout may mean that the call never ran or that it ran and its reply was lost, and summaries drop exact arguments. The execution ledger L_t keeps these apart. Beyond messages and step records, each admitted call becomes an *Operation* record with its exact request, an execution phase (committed, executing, settled), and a settled outcome: succeeded, failed, or uncertain. The ledger projection Π_L shows settled outcomes and recent history in the context, and earlier records, including the code that issued a call, remain retrievable through `task.search`. An uncertain outcome is never re-run by the runtime; whether to verify or retry it is left to the agent, which sees the uncertainty rather than a guess.

3.4 WORK LIFECYCLE (W)

In a transcript-based harness, a commitment such as "report when the repository reaches 2,051 stars" exists only as text: the model must remember it and poll. Perdura makes each commitment an explicit WorkItem $w_j = (\text{goal}_j, \text{done_when}_j, \text{phase}_j, \text{wait}_j)$ that moves from READY to ACTIVE to COMPLETED, or is cancelled. This serves two purposes. First, commitments become explicit state rather than text, visible to all three parties: the agent sees open WorkItems and their waits in its context (Π_W), the runtime does not let a task finish while any WorkItem remains open, and a user can see which commitments are open, which are waiting and on what, and which are done without reading the transcript. Second, work can wait on conditions: an active WorkItem can register a wait on a message, a Handle event, or a deadline and suspend; the scheduler in Q_t tracks the wait and wakes the task when the condition resolves, so no model call occurs while waiting.

3.5 CONTROL STATE (Q)

The namespace, ledger, and work lifecycle describe execution state; Q_t is the part of the runtime that stays active while the model is not. The model runs only inside steps; between steps, Q_t holds the live resources behind Handles (§3.2), such as browser sessions, MCP connections, shell sessions, and in-flight calls, and listens for the events that matter to the task: messages, Handle readiness, subscriptions, and deadlines (§3.4). When a registered condition resolves, Q_t wakes the task and constructs the next context, so the model is invoked when there is something to act on rather than to check whether there is. Every external effect also passes through Q_t: code issues typed syscalls, and Q_t checks their permissions, executes them, and settles their outcomes. It also keeps the task metadata that Π_Q renders as the task definition and callable surface.

Recovery follows from the same role, since Q_t holds the resources and schedules the work that must survive runtime loss. The domains recover separately: after runtime loss, Q_t restores the newest namespace generation that passes its integrity check, falling back to older generations, and rebinds the resources behind recoverable Handles; no historical code is replayed. Unfinished steps are recorded as failed, noting whether their code began, Operations whose outcome cannot be established remain uncertain rather than being re-run, and WorkItem phases change only through agent-issued transitions. Restoring namespace state does not reverse external effects.

4 EXPERIMENTAL EVALUATION

RQ1 examines task outcomes and cost, RQ2 how computational state from one step stays available to later steps, and RQ3 how execution continues when later actions depend on prior evidence or unfinished work.

4.1 SETUP AND TASK COVERAGE

We evaluate across three long-horizon benchmarks with complementary execution demands: SENTINELBENCH (Maldaner et al., 2026) for asynchronous browser monitoring, TOOLMAZE (Zhu et al., 2026) for multi-tool orchestration under failures, and BEAM (Tavakoli et al., 2025) for long-context memory retention. Each harness runs all 100 SENTINELBENCH tasks, 400 TOOLMAZE cases sampled across complexity C1–C4 and perturbation P0–P4, and 400 BEAM questions (80/140/140/40 at 128K/500K/1M/10M tokens); Appendix A details the protocols.

Comparative harnesses and execution environment. We compare Perdura with four agent harnesses: **Claude Code** (Anthropic, 2026), **Codex** (OpenAI, 2026), **Hermes** (Nous Research, 2026), and **OpenClaw** (OpenClaw, 2026). We do not include the research prototypes with persistent execution state discussed in Section 5, as we could not run them end to end on these benchmarks, which require tool, shell, and browser access. To control for base-model differences, all systems use **GLM-5.3 Flash** (Z.ai, 2026) with `low` reasoning effort and default temperature. All harnesses run in identical isolated Docker containers with shared shell endpoints and filesystem sandboxes. For browser tasks, harnesses without built-in browser automation use the official Playwright MCP server (Microsoft, 2026) rather than custom browser wrappers. Every harness receives the same benchmark instructions; for SENTINELBENCH we append one identical sentence to each instruction

Harness	Score (%)	Calls	Input Tokens	Output Tokens	Cost ($)
SentinelBench					
Claude Code	66.00	2,736	103.52M	164.01K	$15.61
Codex	63.00	2,887	81.57M	379.92K	$12.43
Hermes	50.00	4,477	91.92M	459.57K	$14.02
OpenClaw	58.00	2,902	114.20M	176.42K	$17.22
Perdura	63.00	2,514	64.58M	237.98K	$9.81
ToolMaze					
Claude Code	41.50	2,345	42.68M	452.63K	$6.63
Codex	40.75	4,170	55.48M	651.06K	$8.65
Hermes	39.25	2,709	32.24M	393.32K	$5.03
OpenClaw	48.50	2,075	71.75M	1,041.88K	$11.28
Perdura	44.25	3,255	33.69M	501.62K	$5.30
BEAM					
Claude Code	52.29	1,707	61.49M	505.98K	$9.48
Codex	36.80	1,870	33.23M	322.55K	$5.15
Hermes	37.43	2,547	52.80M	506.44K	$8.17
OpenClaw	30.63	2,098	157.00M	671.47K	$23.89
Perdura	51.38	2,780	45.41M	473.67K	$7.05

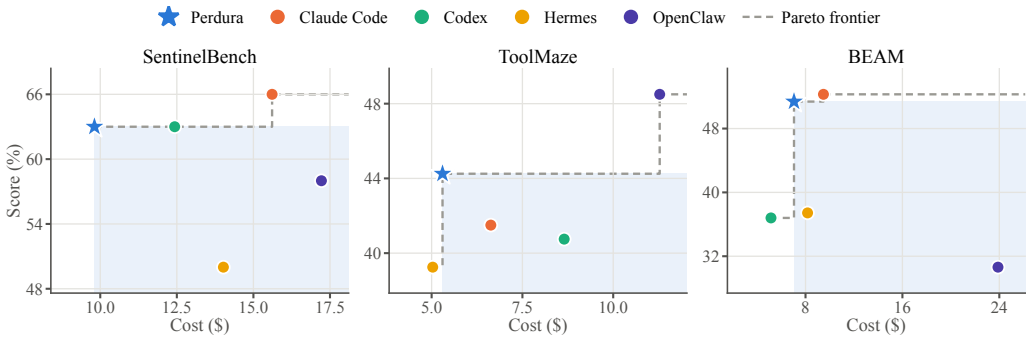


Table 1: **Main evaluation results across three long-horizon benchmarks under GLM-5.3 Flash.** *Top:* task score (%), model calls, input and output tokens, and cost; calls and tokens are counted at the inference gateway for every harness, and cost applies the standard GLM-5.3 Flash price (Appendix A.3). *Bottom:* task score versus cost on each benchmark; the dashed staircase is the score–cost Pareto frontier and the shaded region contains harnesses dominated by Perdura (higher cost and no higher score).

(Appendix A). We evaluate each harness as an integrated system under a shared base model and environment, so end-to-end comparisons measure system-level behavior rather than isolating any single mechanism; Sections 4.3 and 4.4 examine the mechanisms through recorded trajectories.

4.2 End-to-end task outcomes and cost (RQ1)

Table 1 reports task success, model calls, tokens, and cost, with score plotted against cost below it. Perdura's task success is comparable to the top-scoring harness: 63.00 (tied with Codex) versus 66.00 for Claude Code on SentinelBench, 44.25 versus 48.50 for OpenClaw on ToolMaze, and 51.38 versus 52.29 for Claude Code on BEAM. The top-scoring harness costs 1.3–2.1× as much as Perdura, and Perdura uses 26–53% fewer input tokens than it; on SentinelBench, Perdura has the lowest input-token use and cost of all five harnesses. The only harnesses cheaper than Perdura, Hermes on ToolMaze and Codex on BEAM, score 5.0 and 14.6 points lower. No harness therefore achieves both a higher score and a lower cost than Perdura on any benchmark. A paired bootstrap over tasks (10,000 resamples) gives 95% intervals of $[-15, 9]$, $[-8.75, 0]$, and $[-4.7, 3.0]$

(a) N: a 49.7 MB transcript is reused from the namespace — BEAM, 10M-7

```
# step 1: load once and index the turns
text = open(p, encoding='utf-8').read()
turns = [(m.start(), m.group(0)) for m in
         re.finditer(r'^(USER|ASSISTANT):', text, re.M)]
# steps 2-3: reuse, no reload
for idx in [2000, 5000, 9000, ...]:
    pos, tag = turns[idx]; print(text[pos:pos+400])
# model sees: text: str = <<'USER: I'm Richard ...>>  (436 chars)
```

Perdura: 8 calls, **133.6K input tokens**; probe score 0.52.

Claude Code: repeated `grep`/`sed` over files; **60 calls, 4.92M tokens**; 0.37. Others: 0.23–0.28.

(b) L: an argument absent from the context is recovered before a write — TOOLMAZE, C4, P3

```
# step 5
mcp_toolmaze.call('archive_quote_snapshot_native', ...,
                  priority='high')
# step 6: "verify two parameters I can't see"
hist = await task.search(plane='code', query=...)
# -> [step 5] ... priority='high'   => no re-archive
```

Perdura: recovers the argument, **skips a duplicate write**, passes.

Hermes, OpenClaw, Codex: fail the task.

(c) W: a registered wait replaces polling — SENTINELBENCH, microhub-stars

```
# step 3: "alert when it reaches 2,051"
w = await page.wait_for_function("... >= 2051", timeout=0)
await suspend()                          # t = 30 s
# step 5: woken by handle.ready at t = 580 s
snap = await page.aria_snapshot()     # 2,054 stars
```

Perdura: no model call during the 550 s wait; 10 calls, 0.13M input tokens.

Claude Code: 12 fixed browser waits; **32 calls, 1.10M input tokens**.

Figure 3: **Case studies for the three execution-continuity domains.** Code is abridged from recorded Perdura steps; gray comments are added for exposition, except quoted intents, which are the model's own.

points for Perdura's score difference to the top-scoring harness on SENTINELBENCH, TOOLMAZE, and BEAM, none excluding zero, while the top-scoring harness is 1.3–2.1× as costly.

4.3 COMPUTATIONAL-STATE CONTINUITY (RQ2)

We examine recorded trajectories for whether computational state created in one step remains available to later steps without recomputation or full exposure in the model context. Figure 3(a) shows a representative case; Figure 3(b–c) shows the trajectories discussed for RQ3.

Computational-state continuity (N): reuse under a bounded context. In a BEAM ingestion session (conversation 10M-7), the agent must digest a 49.7 MB transcript of 22,652 turns into memory. In step 1 it loads the transcript into `text` and builds a turn index `turns`; steps 2 and 3 sample the conversation's phases directly from these objects, without reloading or rescanning the file. The model sees each object as a single row of about 0.4K characters—its type and a truncated excerpt—while the full values remain available to later code. The ingestion takes 8 model calls and 133.6K input tokens. Claude Code and Codex instead keep this state in files and rescan them with shell tools whose outputs enter the context; Claude Code uses 60 calls and 4.92M input tokens, and OpenClaw 142 calls and 26.7M. Probes on this conversation score 0.52 on average for Perdura versus 0.23–0.37 for the others. This comparison is illustrative: it shows where the intermediate state lives, not a controlled effect of N, and the probe scores also depend on the memory the ingestion writes.

4.4 EXECUTION CONTINUITY (RQ3)

We next examine two forms of cross-step dependency represented explicitly by Perdura: prior execution evidence through L (requirement R3) and unfinished work through W (requirement R4).

Execution-evidence continuity (L): recovering earlier call arguments. In the TOOLMAZE task, the agent archives a quote snapshot with `priority='high'` at step 5. At step 6, before its final actions, it needs to confirm that argument, but its context contains only the step-5 intent and outputs,

and the archive receipt does not echo the priority. The agent queries the code plane of the ledger, which returns the step-5 call with its exact arguments. The recorded arguments decide the branch: the archive is not repeated, and the recommendation call is confirmed to have used the requested category. Without the recorded call, the agent would have to either trust its own recollection or repeat a write whose earlier effect it cannot see.

Unfinished-work continuity (W): waiting without polling. The SENTINELBENCH task requires reporting once a repository has gained 200 stars, which happens several minutes into the episode. After reading the initial count of 1,851, the agent expresses the target as a page-side predicate, registers it as a wait on the page handle, and suspends. The runtime issues no model call for 550 s; when the predicate holds, the wait's result arrives as a `handle.ready` message, and the agent reads the new count from the same page and submits the report. In the compared Claude Code trajectory, the agent instead polls with twelve fixed-duration browser waits, each followed by a model turn to re-inspect the page. Under polling, the number of model calls grows with the wait duration divided by the polling interval; a registered wait requires one call to establish it and one after it resolves, independent of the wait duration. Across all SENTINELBENCH tasks, Perdura spends 63% of task time suspended, with no model call while suspended, and has a model call in flight for only 31% of the time. Claude Code and Codex spend 29% and 39% of task time in wait or sleep tool calls, mostly in 30 s slices, and have a call in flight for 54% and 57%; after their waits they issue 92 and 108 re-check calls per waiting hour, about three times Perdura's 33 resumptions. Hermes and OpenClaw traces do not record wait durations reliably and are omitted.

5 RELATED WORK

Code execution and persistent computation. CodeAct uses executable Python actions and interpreter feedback (Wang et al., 2024). TaskWeaver retains execution state across turns (Qiao et al., 2023), while CaveAgent pairs a semantic stream with persistent Python objects (Ran et al., 2026). The benefit of persistence also depends on model behavior: for one task family and base model, May et al. (2026) find that training–runtime mismatches affect token use and execution stability, without a statistically significant difference in solution quality.

Externalized execution state. Ledger tracks observed, modified, and attempted state to inform the model and govern commands (Wang et al., 2026b). OpenRath places conversation, lineage, tool evidence, and pending work in a Session with fork, merge, and replay operations (Wen et al., 2026). Scroll combines an event log and persistent Python namespace with access to history evicted from the working context (Lin et al., 2026). Other systems make progression or recovery explicit through workflow transitions in StateFlow (Wu et al., 2024), MAGE's branching subgoal tree (Chen et al., 2026), replayable session logs in the OpenHands Software Agent SDK (Wang et al., 2026a), and persistent processes and ambiguous-outcome recovery in Agent libOS (Zhang, 2026).

Positioning. Each form of state that Perdura manages has precedents in the systems above; Perdura's contribution is to keep computational state, execution evidence, and unfinished work separate within one runtime, because they fail and recover differently (Section 3.5).

6 CONCLUSION

We presented Perdura, a stateful execution harness in which the runtime owns execution continuity: it separates a persistent namespace (N), an execution ledger (L), and a work lifecycle (W), and gives the model a bounded context. Across three benchmarks under GLM-5.3 Flash, Perdura attains task success comparable to the strongest harness at 47–74% of its cost. Case studies show how namespace reuse, recorded execution evidence, and registered waits let execution continue without reconstructing state from the interaction history.

Statement on the Use of AI

In accordance with the ICLR policy on generative AI, we disclose that large language models and generative AI tools were used during the preparation of this work for: (1) **Writing and exposition assistance**, including drafting sections of the text for the authors to revise, proofreading, and polishing text for grammatical precision and readability; (2) **Code implementation**, specifically assisting in writing auxiliary benchmark harness adapters, evaluation scripts, and tooling utilities; (3) **Literature survey**, aiding in querying and structuring related work references and verifying citations; and (4) **Data analysis**, assisting in computing statistics from logged experiment results and auditing run validity; all reported numbers were produced by scripts and checked by the authors. Generative AI tools were not used for formulating research hypotheses, theoretical derivations, or synthetic data generation. All AI-assisted code was manually reviewed, debugged, and verified through execution tests, and all text and citations were checked against original sources. The authors take full responsibility for the integrity, correctness, and conclusions of this manuscript.

Reproducibility Statement

Appendix A specifies the benchmark subsets and protocols, the execution environment and model configuration shared by all harnesses, and how calls, tokens, and cost are counted.

REFERENCES

Anthropic. Claude code documentation, 2026. URL https://code.claude.com/docs/en/.

Yaoqi Chen, Haibin Lai, Yuru Feng, Chuyu Han, Qianxi Zhang, Baotong Lu, Menghao Li, Xinjiang Wang, Zhirui Wang, Shusen Xu, Zengzhong Li, Zewen Jin, Hao Wu, Cheng Li, and Qi Chen. Beyond semantic organization: Memory as execution state management for long-horizon agents. *arXiv preprint arXiv:2606.06090*, 2026. URL https://arxiv.org/abs/2606.06090.

Guanting Dong, Xiaoshuai Song, Yuyang Hu, Jiajie Jin, Chenghao Zhang, Yifei Chen, Xiaoxi Li, Huaying Yuan, Xinyu Yang, Tongyu Wen, Jiejun Tan, Hongjin Qian, Shijue Huang, Junting Lu, Zhenyu Li, Wanjun Zhong, Yutao Zhu, Tat-Seng Chua, Zhicheng Dou, and Ji-Rong Wen. Towards long-horizon agents: A survey. Preprints.org, version 1, 2026. URL https://www.preprints.org/manuscript/202607.1328/v1.

Yin Lin, Elaine Ang, Erkang Zhu, Bolin Ding, and Jingren Zhou. Context as an environment: Programmatic context management for long-horizon agents. arXiv:2608.21690, 2026. URL https://arxiv.org/abs/2608.21690.

Matheus Kunzler Maldaner, Adam Fourney, Amanda Swearngin, Hussein Mozannar, Gagan Bansal, Maya Murad, Rafah Hosn, and Saleema Amershi. Sentinelbench: A benchmark for long-running monitoring agents. arXiv preprint arXiv:2606.05342, 2026. URL https://arxiv.org/abs/2606.05342.

Victor May, Aaditya Salgarkar, Yishan Wang, Diganta Misra, and Huu Nguyen. Agents learn their runtime: Interpreter persistence as training-time semantics. *arXiv preprint arXiv:2603.01209*, 2026. URL https://arxiv.org/abs/2603.01209.

Qianyu Meng, Yanan Wang, Liyi Chen, Wei Wu, Yihang Li, Wenyuan Jiang, Qimeng Wang, Chengqiang Lu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. Preprints.org, version 3, 2026. URL https://www.preprints.org/manuscript/202604.0428/v3.

Microsoft. Playwright MCP, 2026. URL https://github.com/microsoft/playwright-mcp.

Nous Research. Hermes agent, 2026. URL https://github.com/NousResearch/hermes-agent.

OpenAI. Codex: Lightweight coding agent that runs in your terminal, 2026. URL https://github.com/openai/codex.

OpenClaw. Openclaw documentation, 2026. URL https://docs.openclaw.ai/.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. arXiv:2310.08560, 2023. URL https://arxiv.org/abs/2310.08560.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, Minghua Ma, Pu Zhao, Si Qin, Xiaoting Qin, Chao Du, Yong Xu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. TaskWeaver: A code-first agent framework. arXiv:2311.17541, 2023. URL https://arxiv.org/abs/2311.17541.

Maohao Ran, Zhenglin Wan, Cooper Lin, Yanting Zhang, Hongyu Xin, Hongwei Fan, Yibo Xu, Beier Luo, Yaxin Zhou, Wangbo Zhao, Lijie Yang, Lang Feng, Fuchao Yang, Jingxuan Wu, Yiqiao Huang, Chendong Ma, Yusen Huang, Dailing Jiang, Jianbo Deng, Sirui Han, Yang You, Bo An, Yike Guo, and Jun Song. CaveAgent: Transforming LLMs into stateful runtime operators. *arXiv preprint arXiv:2601.01569*, 2026. URL https://arxiv.org/abs/2601.01569v4.

Mohammad Tavakoli, Alireza Salemi, Carrie Ye, Mohamed Abdalla, Hamed Zamani, and J Ross Mitchell. Beyond a million tokens: Benchmarking and enhancing long-term memory in LLMs. arXiv preprint arXiv:2510.27246, 2025. URL https://arxiv.org/abs/2510.27246.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang H. Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The OpenHands software agent SDK: A composable and extensible foundation for production agents. In *Proceedings of Machine Learning and Systems*, volume 8, 2026a. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/8ae9cf363ea625161f885b798c1f1f78-Abstract-Conference.html.

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:2608.00808, 2026b. URL https://arxiv.org/abs/2608.00808.

Fukang Wen, Zhijie Wang, and Ruilin Xu. OpenRath: Session-centered runtime state for agent systems. *arXiv preprint arXiv:2606.19409*, 2026. URL https://arxiv.org/abs/2606.19409.

Yiran Wu, Tianwei Yue, Shaokun Zhang, Chi Wang, and Qingyun Wu. StateFlow: Enhancing LLM task-solving through state-driven workflows. In *Proceedings of the First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=3nTbuygoop.

Z.ai. GLM-5.3-flash: More intelligence with less compute, 2026. URL https://autoclaw.z.ai/blog/model/glm-5.3-flash/.

Yingqi Zhang. Agent libOS: A runtime substrate for capability-controlled self-evolving LLM agents. *arXiv preprint arXiv:2606.03895*, 2026. URL https://arxiv.org/abs/2606.03895.

Dongsheng Zhu, Xuchen Ma, Yucheng Shen, Xiang Li, Yukun Zhao, Shuaiqiang Wang, Lingyong Yan, and Dawei Yin. When tools fail: Benchmarking dynamic replanning and anomaly recovery in LLM agents. arXiv preprint arXiv:2606.05806, 2026. URL https://arxiv.org/abs/2606.05806.

A Experimental Protocol and Execution Environment

A.1 Benchmark Specifications and Evaluation Protocols

The evaluation covers three benchmarks, 900 task instances per harness:

- **SentinelBench** (Maldaner et al., 2026): All 100 official tasks on asynchronous monitoring and browser interaction. Following the official protocol, the 630-second limit (a 600 s simulation window plus a 30 s reaction window) starts when the agent first opens the task URL. We append one sentence to every task instruction, identical across all harnesses, stating that environment changes occur asynchronously within a 10-minute window; task content and scoring are otherwise unchanged.
- **ToolMaze** (Zhu et al., 2026): Multi-tool orchestration; all harnesses reach the ToolMaze tools through a Model Context Protocol (MCP) server. We sample 400 cases uniformly from the official distribution across complexity tiers C1–C4 and perturbation tiers P0–P4: P0 is unperturbed, and P1–P4 inject explicit tool errors (e.g., HTTP-style failures) or implicit, semantically incorrect outputs, each transient or permanent. All harnesses run with guidance (hint) mode disabled.
- **BEAM** (Tavakoli et al., 2025): Long-context memory and retrieval at four context lengths (80 questions at 128K tokens, 140 at 500K, 140 at 1M, and 40 at 10M; 400 questions, stratified by context length). BEAM presents the full conversation before its probing questions; we adapt this by running ingestion and probing as separate sessions of each harness, because harnesses carry dialogue history differently. Starting the probe session fresh ensures that every harness answers from state it retained during ingestion rather than from an in-context transcript. The protocol is identical across harnesses; each harness runs with its default configuration and may rely on whatever state it chooses to retain, except that OpenClaw runs without its embedding-based memory.

A.2 Execution Sandboxing and Comparative Harnesses

All harnesses run under the same execution conditions:

- **Sandboxing and Tool Interfaces**: Every harness runs in an isolated Docker container with the same system dependencies, shell environment, and filesystem boundaries. For browser tasks, harnesses without a built-in browser controller use the official Playwright MCP server (Microsoft, 2026).
- **Model Configuration**: All comparative harnesses (**Claude Code** (Anthropic, 2026), **Codex** (OpenAI, 2026), **Hermes** (Nous Research, 2026), and **OpenClaw** (OpenClaw, 2026)) and Perdura are driven by **GLM-5.3 Flash** (Z.ai, 2026). We use `low` reasoning effort and the provider-default temperature in all runs. Every harness, including Perdura, runs with its default configuration; no benchmark-specific prompts or tool descriptions are added.

A.3 Resource Accounting Methodology

In Table 1, input tokens count all prompt tokens over every model call in a task, including cached prompt tokens; output tokens count all generated tokens. Cost applies the standard GLM-5.3 Flash price to these counts. Tokens used by benchmark simulators and judges are excluded.