TORCH: JOINTLY EVOLVING TASK EXECUTION AND ONLINE SUPERVISION

Anonymous authors
Paper under double-blind review

ABSTRACT

Language-model agents increasingly perform complex tasks through sustained interactions with tools and environments. During execution, agents may repeat ineffective actions, overlook contradictory evidence, or terminate before adequately verifying results. Existing methods primarily use post-hoc reflection to improve subsequent attempts or online supervision to correct ongoing execution, with some further retaining skills and memory for experience reuse. However, task execution and supervisory assessment involve distinct decisions, and how to accumulate their respective experience from the same interaction so that both roles can continually improve remains an open question. In this paper, we propose TORCH, a framework that corrects execution trajectories through online supervision and supports separate experience accumulation for the worker and supervisor. Its core idea is to maintain independent long-term memories for the two roles, enabling the worker to learn task-solving knowledge and the supervisor to learn diagnostic and guidance strategies, so that both roles can continually evolve while model parameters remain fixed. Further, we introduce TORCH-GA, a harness built on GenericAgent that fully implements the TORCH framework. We evaluate runtime supervision on external harnesses and memory evolution and transfer in TORCH-GA. Empirical results demonstrate that TORCH improves task performance, that joint evolution achieves higher mean later-round performance than either one-sided variant across three independent ten-round runs per condition, and that accumulated experience remains useful across models and task domains without further memory updates. Our code is available at `https://anonymous.4open.science/r/TORCH-D8FF/`.

1 INTRODUCTION

Language-model agents increasingly undertake complex, multi-step tasks through sustained tool use, environment interaction, and feedback interpretation (Yao et al., 2023; Wang et al., 2024b; Yang et al., 2024; Wang et al., 2025). However, successful individual actions do not always ensure correct task completion (Jimenez et al., 2024; Merrill et al., 2026; Xu et al., 2026). During extended interactions, agents may repeat ineffective actions, overlook evidence that contradicts their current assessment, or terminate before adequately verifying the result (Laban et al., 2026; Zhu et al., 2025; Liu, 2026). Reliable execution therefore requires both choosing the next action and recognizing when the current process has deviated from the task objective.

Existing research addresses these problems primarily through post-hoc reflection and online supervision. Post-hoc reflection extracts lessons from an attempt for subsequent attempts (Madaan et al., 2023; Shinn et al., 2023), whereas online supervision inspects ongoing execution and provides corrective guidance (Lin et al., 2026a; Liu et al., 2026). Some methods further combine supervision with persistent skills and memory for experience reuse (Xiao et al., 2026). However, how experience is accumulated and reused raises two related issues. First, existing approaches distinguish execution from supervision at runtime but do not sufficiently distinguish the two roles as separate targets of learning and evolution. Learning how to solve a task does not directly teach a supervisor when to intervene or what guidance to provide. Without separately accumulating supervisory experience, improvements in task execution may not translate into more effective supervision. Second, it remains unclear whether the worker and supervisor can reuse their learned memories with a different model or on different tasks, without updating those memories.

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.

Achieving both role-specific learning and experience reuse is challenging: the same interaction provides different learning evidence for the two roles, and that evidence is intertwined with details of the original task. The worker's record contains its actions, environment feedback, and received guidance, providing evidence about how to act, debug, and verify results. The supervisor's record additionally contains the inspections and reasoning behind an intervention. Meanwhile, guidance changes subsequent execution, and new execution evidence informs the next supervisory assessment. Each role must therefore review its own decisions alongside the shared task result to determine what to retain in its memory. For these memories to remain useful beyond the original tasks, they must also capture reusable procedures and diagnostic principles rather than merely reproduce a particular execution trace.

To address these challenges, we propose **Trajectory Oversight and Runtime Correction Harness (TORCH)**, a framework for online supervision and continual experience accumulation. Its core idea is to correct execution through worker–supervisor interaction and organize learning around each role's own records and long-term memory. To fully realize TORCH, we design TORCH-GA, a harness built on GenericAgent (Liang et al., 2026) specifically for the complete execution and learning workflow. It integrates separate worker and supervisor records, post-evaluation reflection, and round-end memory merging, enabling both roles to accumulate and reuse experience while model parameters remain fixed. For external harnesses, we retain the original worker and attach a same-model TORCH supervisor with frozen supervisory memory. The memories accumulated in TORCH-GA can also be frozen and supplied to other model or task configurations, enabling evaluation of experience reuse without target-task updates.

We evaluate TORCH supervisors across harnesses and study memory evolution and transfer in TORCH-GA. Runtime supervision improves task performance across the evaluated harnesses. Across three independent ten-round TORCH-GA runs per condition, joint evolution achieves higher mean later-round performance than either one-sided variant. Frozen-memory transfer improves performance across models and benchmarks, including software feature development on FeatureBench (Zhou et al., 2026) and office-artifact and web-page tasks on WorkBuddy Bench (Tencent WorkBuddy Bench Team et al., 2026).

Our contributions are summarized as follows.

- We propose TORCH, a framework that connects same-model online supervision with role-specific experience accumulation, with a supervision mechanism applicable across different worker harnesses.
- We design TORCH-GA, a harness built on the open-source GenericAgent specifically to fully implement TORCH, integrating role-specific interaction records, post-evaluation reflection, and memory merging for continual worker and supervisor evolution.
- We evaluate runtime supervision across harnesses, compare joint evolution with both one-sided variants, and demonstrate the reuse of frozen role-specific memories across models and task domains.

2 Related Work

2.1 Post-hoc Reflection

Post-hoc reflection improves subsequent attempts using feedback. Self-Refine uses model-generated critiques, Self-Debugging combines code explanation with execution feedback, and CRITIC checks results with external tools (Madaan et al., 2023; Chen et al., 2024; Gou et al., 2024). Reflexion retains verbal reflections, while ExpeL extracts reusable insights (Shinn et al., 2023; Zhao et al., 2024). Self-correction without reliable external feedback remains limited (Huang et al., 2024). TORCH instead uses current execution evidence to correct actions within the ongoing task.

2.2 Online Agent Supervision

Multi-agent frameworks coordinate specialized roles through conversation or SOP-style workflows (Wu et al., 2023; Hong et al., 2024). Within this division of labor, supervision assesses ongoing execution and guides subsequent actions. Related work studies intermediate-step assessment

and feedback-based correction (Lightman et al., 2024; Wang et al., 2024a; Zhou et al., 2024). For tool-using agents, SupervisorAgent provides modular runtime guidance, while Shepherd supports programmable supervision through reversible execution traces (Lin et al., 2026a; Yu et al., 2026). LivePlan triggers advice through rule-based monitoring, and Steer, Don't Solve trains a small critic to guide a frozen coding agent (Liu et al., 2026; Gandhi et al., 2026). PILOT combines same-model steering with live skill and memory updates (Xiao et al., 2026); LoopArena evaluates runtime controllers for iterative execution (Wang et al., 2026). TORCH's same-model supervisor attaches to existing harnesses while preserving the worker's execution loop, tools, and responsibility for final submission.

2.3 Experience Reuse and Harness Evolution

Beyond correcting ongoing execution, agents can improve future behavior by adapting their harnesses or retaining experience. Meta-Harness, Agentic Harness Engineering, and Self-Harness evolve harness code or components, while Continual Harness studies adaptation during a continuous run (Lee et al., 2026; Lin et al., 2026b; Zhang et al., 2026; Karten et al., 2026). Other agents retain experience through memory, reflection, and skill libraries (Park et al., 2023; Wang et al., 2023; Packer et al., 2024). PILOT writes skills and memory during execution and retains updates from successful runs (Xiao et al., 2026). Evolved harnesses have also been reused across models or tasks (Lee et al., 2026; Lin et al., 2026b). Failure traces also provide localized diagnostic evidence (Luan et al., 2026). TORCH separately accumulates task-solving and supervisory experience through post-evaluation reflection on each role's records. Its evidence review retains supported failure patterns and verification lessons from unsuccessful runs. We evaluate reuse by transferring the resulting memories without further updates.

3 Method

TORCH is a framework combining online supervision, role-specific reflection, and memory evolution. TORCH-GA is a harness built on GenericAgent that fully implements this framework.

3.1 Problem Setting

For task x, the worker trajectory after k actions is $\tau_k = (a_0, o_1, \ldots, a_{k-1}, o_k)$, where a_i is an action and o_{i+1} its observed response. The worker and supervisor share a frozen model θ but maintain independent contexts. The worker acts and submits the result; the supervisor inspects evidence and provides guidance.

TORCH maintains role-specific memories $M_t = (M_t^W, M_t^S)$: M_t^W stores task-solving, debugging, and verification experience; M_t^S stores failure patterns, diagnostic evidence, and intervention strategies. We use M_0 for initial memory templates without evolved experience. In full-benchmark evolution, round r uses M_{r-1} and produces M_r after evaluation and merging; Section 4.3 defines local indexing for the 40-task continuation study. Model parameters, prompts, tools, and runtime controls remain fixed during evolution.

3.2 Methodology Motivation and Overall Framework

A shared interaction can teach the two roles different lessons. In the webserver case (Appendix C), a command exited successfully but HTTP returned no expected content. Supervisor-guided checks and a failed restart exposed a surviving server process; the worker repaired the service and verified the content. The worker can learn repair and verification procedures, while the supervisor can learn to question apparent success and refine diagnosis when earlier guidance fails. Separate records preserve the evidence needed for each role's learning. A corrective message tells the worker what to check or change, but does not preserve the full diagnostic process that produced it. Learning only from the worker trajectory would therefore omit evidence about how the supervisor assessed the situation. The worker's own record captures how it carries out the repair and verifies the result.

TORCH comprises three stages (Figure 1): (1) **same-model runtime supervision** produces corrective guidance and separate interaction records; (2) **role-specific post-evaluation reflection** extracts candidate memory updates from each role's records and the shared task evaluation result; (3)

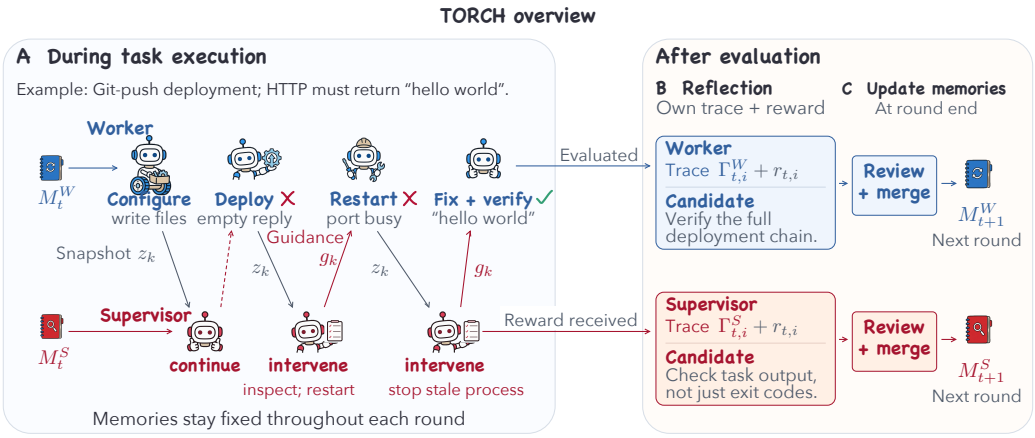

Figure 1: Overview of the TORCH framework. (A) An illustrative webserver sequence shows repeated runtime reviews: the supervisor reads snapshots z_k and either lets execution continue without a message or sends guidance g_k; the worker performs and verifies repairs. (B) Once the task reward $r_{t,i}$ is available, both roles reflect on their own records $\Gamma_{t,i}^R$ using this shared reward to produce candidate memory updates. (C) Candidates are reviewed and merged into each role's own memory at round end, producing M_{t+1}^W and M_{t+1}^S for the next round. Both roles use the same frozen model with separate contexts; execution memories remain fixed within each round.

evidence-guided memory merging reviews candidates and forms separate memories for the next round.

To support this workflow across successive rounds, we design TORCH-GA, a harness built on GenericAgent specifically to fully implement TORCH. It preserves each role's interaction history, restores that context for post-evaluation reflection, and passes candidate updates to separate memory-merging processes. Both roles execute with fixed memories within a round and receive reviewed updates in the next. The runtime supervision interface can also attach a supervisor with frozen memory to external harnesses, which retain their original workers without adopting TORCH's worker-memory or evolution mechanisms.

3.3 Same-Model Runtime Supervision

At review point $k \in \mathcal{K}$, the worker pauses and its harness adapter constructs a snapshot

$$z_k = \Phi_H(x, \tau_k, D_{<k}),$$

containing the task, observed execution, runtime state, and earlier supervisory decisions $D_{<k}$. Here, $\mathcal{K}$ denotes the set of review-point indices, H denotes the worker harness, and Φ_H is its snapshot-construction function. The supervisor inspects this evidence in an independent context, retaining prior reviews and accessing frozen memory M_t^S. It can inspect task artifacts and run diagnostics but cannot modify existing task artifacts or submit results. It returns

$$d_k \in \{\texttt{continue}, \texttt{intervene}(g_k)\}.$$

`continue` resumes execution unchanged; `intervene` appends guidance g_k as a user message before resuming the worker's original loop. At attempted completion, the supervisor again chooses between `continue` and `intervene`; the worker performs any further actions and submits the result. TORCH-GA reviews periodically, after stalls and interventions, and at attempted completion. Default snapshots omit worker reasoning; on-demand retrieval exposes the complete recorded trajectory. Appendix A details triggers, observations, diagnostic permissions, and correction controls.

3.4 Role-Specific Post-Evaluation Reflection

For execution i in a round starting from M_t, let $\Gamma_{t,i}^W$ and $\Gamma_{t,i}^S$ denote the role-visible records. The worker record contains actions, feedback, and received guidance, but not the supervisor's internal

diagnosis. The supervisor record preserves its review contexts, inspections, diagnostic interactions, decisions, and completion review. Once evaluation of the worker's submitted result returns the scalar task reward $r_{t,i}$, both roles reflect on their own interaction records using this same reward. Restoring the saved conversation lets reflection use the context in which the role made its decisions. Hidden tests and grading criteria remain inaccessible during both execution and reflection (Appendix B.1). The post-evaluation reflection procedure $\mathcal{R}_R$ produces a candidate memory state for role R:

$$\widehat{M}^R_{t,i} = \mathcal{R}_R\left(\Gamma^R_{t,i}, r_{t,i}; M^R_t\right), \qquad R \in \{W, S\}.$$

The same procedure produces reflection text $F^R_{t,i}$, describing proposed changes and supporting observations. The candidate, reward, and reflection form $C^R_{t,i} = (\widehat{M}^R_{t,i}, r_{t,i}, F^R_{t,i})$ for merge review. The merge reviewer uses the task reward and the role's reflection to assess the proposed memory changes before accepting them.

3.5 Evidence-Guided Memory Merging

Task success does not validate every action or suggestion: a task may succeed even when supervisory advice was never followed. Conversely, failed executions can reveal useful failure patterns or missing verification steps. Merge review therefore distinguishes observed facts, failure boundaries, and unverified hypotheses before keeping, combining, rewriting, or rejecting candidates. Unsupported suggestions remain unverified; conflicting advice is rewritten as a neutral verification principle or rejected when evidence cannot resolve it.

Both memory banks remain fixed throughout a round. After its N_t task executions and reflections finish, the round-end update procedure $\mathcal{U}_R$ merges, reviews, and validates each role's candidates:

$$M^R_{t+1} = \mathcal{U}_R\left(M^R_t, C^R_{t,1}, \ldots, C^R_{t,N_t}\right), \qquad R \in \{W, S\}.$$

For each role, candidates are processed in a fixed execution order. Each candidate is merged by comparing it with the round-start memory and the memory updated by earlier candidates. The reviewer checks every modified memory file against the task reward and reflection, even when there is no textual conflict. Validation checks that edits stay within the permitted files, every modified file has a recorded review decision, and the memory structure is valid and free of conflict markers. Failed checks trigger revision. Accepted updates become available in the next round (Appendix B).

The operators do not exchange candidate memory text. The roles instead influence each other's learning through execution: guidance changes the worker trajectory, and new execution evidence informs later reviews. Joint evolution couples these interactions with continuing updates to both memories.

4 Experiments

We test runtime supervision across harnesses, role-selective memory evolution, and frozen-memory transfer.

4.1 Experimental Setup

Benchmarks and metrics. We evaluate all 89 Terminal-Bench 2.1 tasks and 500 SWE-bench Verified tasks (Merrill et al., 2026; Jimenez et al., 2024; Bin et al., 2026) using pass@1 with programmatic verification and one execution per task per configuration or evolution round. Within-domain transfer uses 23 CPU tasks from FeatureBench-lite (Zhou et al., 2026) (Appendix E). WorkBuddy Bench (Tencent WorkBuddy Bench Team et al., 2026) transfer covers 50 Office tasks (spreadsheets, documents, and presentations) and 70 Web tasks (long-horizon front-end development). All three transfer sets use three executions and evaluations per task (avg@3), reporting task pass rates for FeatureBench and partial-credit rubric scores for Office/Web. We report the mean and sample standard deviation across three benchmark-level repeat scores. Repeats also account for LLM-judge variability in Office/Web; Claude Opus 4.8 judges visual criteria. Execution errors remain in the denominator.

Table 1: Coding-agent performance (pass@1, %). Selected references follow system-card results for (a) (Anthropic, 2026) and Vals AI evaluations for (b) (Vals AI, 2026); see Appendix A for settings and scope. Lower blocks use Qwen3.8-27B-FP8 workers with native supervision systems (Others) or TORCH runtime supervision across harnesses (Ours).

(a) Terminal-Bench 2.1

Model (Selected)	Harness	pass@1
Claude Opus 4.8	Terminus-2	74.6
Claude Opus 4.7	Terminus-2	66.1
GPT-5.5	Terminus-2	**78.2**
Gemini 3.1 Pro	Terminus-2	70.3

Qwen3.8-27B-FP8 (Others)	Supervisor	
System	off	on
SupervisorAgent	31.5	39.3
Steer, Don't Solve	42.7	40.4
LivePlan	39.3	49.4

Qwen3.8-27B-FP8 (Ours)	Supervisor	
Harness	off	on
Terminus-2	65.2	77.5
Claude Code	61.8	80.9
OpenCode	66.3	79.8
TORCH-GA	74.2	**85.4**

(b) SWE-bench Verified

Model (Selected)	Harness	pass@1
Claude Opus 4.8	mini-SWE-agent	**88.6**
Claude Opus 4.7	mini-SWE-agent	82.0
GPT-5.5	mini-SWE-agent	82.6
Gemini 3.1 Pro	mini-SWE-agent	78.8

Qwen3.8-27B-FP8 (Others)	Supervisor	
System	off	on
SupervisorAgent	67.6	67.4
Steer, Don't Solve	69.2	67.6
LivePlan	66.6	72.6

Qwen3.8-27B-FP8 (Ours)	Supervisor	
Harness	off	on
mini-SWE-agent	73.8	82.0
Claude Code	80.4	87.4
OpenCode	83.2	86.8
TORCH-GA	73.0	**89.2**

Models and comparison protocol. Main experiments use Qwen3.8-27B-FP8; cross-model transfer uses GLM-5.3 and DeepSeek-V4.1-Flash. TORCH supervisors use the same frozen model as the worker in a separate context. Off/on comparisons fix the worker model, tools, and harness. External harnesses retain their original workers without TORCH's worker-memory mechanism; on attaches a supervisor with frozen supervisory memory. TORCH-GA comparisons additionally fix worker memory M_t^W; on activates the supervisor with M_t^S. All memory-evolution and transfer experiments use TORCH-GA. Appendix D reports recorded token usage, including both roles when supervision is on.

4.2 Runtime Supervision Across Harnesses

Table 1 reports complete-system scores alongside published closed-source references and measures supervision through within-configuration off/on comparisons. TORCH-GA evolves both memories from initial templates separately for each benchmark: 20 rounds on all 89 Terminal-Bench 2.1 tasks and 5 rounds on all 500 SWE-bench Verified tasks. Its rows report final-round execution with M_{19} and M_4, respectively, retaining evolved worker memory in both conditions. Supervisors attached to external harnesses use the same frozen supervisor memory as the corresponding TORCH-GA evaluation. Published reference scores follow their original protocols.

Comparison with existing supervision systems. PILOT (Xiao et al., 2026) has no public implementation at submission; we compare against the authors' implementations of SupervisorAgent, Steer, Don't Solve, and LivePlan (Lin et al., 2026a; Gandhi et al., 2026; Liu et al., 2026) with Qwen3.8-27B-FP8 workers. SupervisorAgent uses a Claude Sonnet 4.6 supervisor; Steer uses a trained 4B critic with mini-swe-agent; LivePlan uses a rule monitor and GPT-5.3-Codex advisor with SWE-agent. Harnesses, supervisory models, and policies differ; Appendix A.2 gives versions and switches.

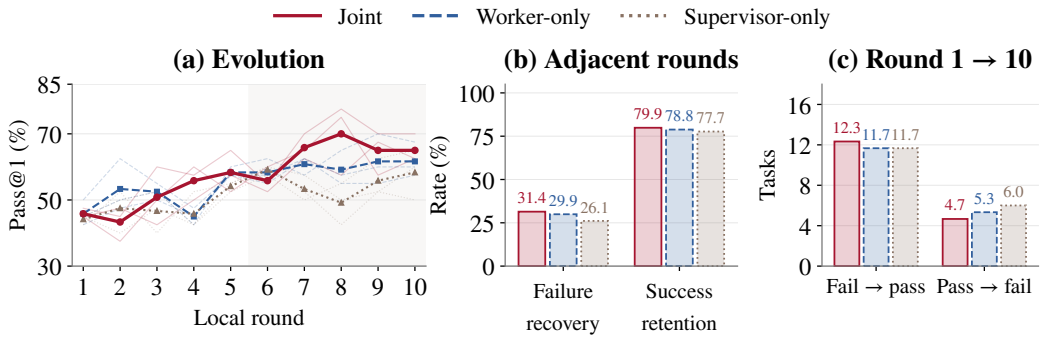

Figure 2: Performance and task transitions during TORCH-GA memory evolution on 40 Terminal-Bench tasks, with three independent ten-round runs per condition starting from M_5 and supervision active throughout. (a) Thin lines show three independent runs per condition; bold lines show mean pass@1. Shading marks rounds 6–10. (b) Failure recovery and success retention across adjacent rounds. (c) Fail-to-pass and pass-to-fail counts from round 1 to 10. Bars average run-level statistics; definitions and SDs appear in Appendix D.4.

4.3 Self-Evolution of TORCH-GA

To compare joint and one-sided learning, we study 40 Terminal-Bench 2.1 tasks with at least one failure in five full-benchmark rounds. We perform three independent ten-round runs per condition, each initialized from the same memory pair produced after round 5, with supervision active throughout. Joint evolution updates both roles; each one-sided variant updates only the named role and freezes the other at this shared starting snapshot. Memories remain fixed within each round. With local round indexing, round r uses $\widetilde{M}_{r-1}$, where $\widetilde{M}_0 = M_5$. Each run independently generates its subsequent reflections and memory updates.

Measurements across runs. We compute performance and transition statistics within each run, then report means and sample standard deviations across three runs (Figure 2; Appendix D.4).

4.4 Transfer of Evolved Memories

Transfer uses M_{20}, produced after Terminal-Bench round-20 evaluation and merging with Qwen3.8-27B-FP8, denoted M_{Qwen} or M_{TB} by source model or benchmark. The worker uses either its initial memory template (Initial) or its transferred memory. The supervisor is disabled (Off), enabled with its own initial memory template (Initial), or enabled with its own transferred memory. Initial templates contain no evolved experience. All memories remain fixed during evaluation, with no updates from target-task feedback.

Across model families. Both roles switch to GLM-5.3 or DeepSeek-V4.1-Flash. Table 2 evaluates all 89 Terminal-Bench tasks, combining initial or transferred worker memory with supervision disabled, initial supervisor memory, or transferred supervisor memory.

Across benchmarks. Table 3 groups targets by task domain: software feature development (in-domain) and office artifacts/web pages (out-of-domain), with Qwen3.8-27B-FP8.

5 Analysis

5.1 TORCH Supervision Improves Execution Across Harnesses

TORCH supervision improved all eight harness–benchmark pairs, with gains of 11.2–19.1 percentage points on Terminal-Bench and 3.6–16.2 on SWE-bench Verified (Table 1). TORCH-GA achieved the highest supervised scores among the evaluated TORCH configurations: 85.4% and

Table 2: Cross-model memory transfer with TORCH-GA on Terminal-Bench 2.1 (pass@1, %; 89 tasks). $M_{\text{Qwen}} = M_{20}$. Initial denotes initial memory; Off disables supervision. Bold marks the highest score for each model.

GLM-5.3		
Worker memory	**Supervisor**	**Pass@1**
Initial	Off	73.0
	Initial	78.7
	M^S_{Qwen}	79.8
M^W_{Qwen}	Off	83.2
	Initial	82.0
	M^S_{Qwen}	**88.8**

DeepSeek-V4.1-Flash		
Worker memory	**Supervisor**	**Pass@1**
Initial	Off	85.4
	Initial	85.4
	M^S_{Qwen}	84.3
M^W_{Qwen}	Off	89.9
	Initial	91.0
	M^S_{Qwen}	**93.3**

Table 3: Cross-benchmark transfer with TORCH-GA (avg@3, %; mean ± SD): task pass rates for FeatureBench (23 CPU tasks), rubric scores for Office/Web (50/70 tasks). $M_{\text{TB}} = M_{20}$ is frozen. Initial denotes initial memory; Off disables supervision. Best means are bold.

Model	**Worker memory**	**Supervisor**	**In-domain**	**Out-of-domain**	
			FeatureBench	**Office**	**Web**
Qwen3.8-27B-FP8	Initial	Off	$13.04_{\pm 4.35}$	$71.60_{\pm 1.32}$	$75.40_{\pm 2.06}$
		Initial	$26.09_{\pm 4.35}$	$73.71_{\pm 1.24}$	$75.71_{\pm 2.73}$
		M^S_{TB}	$27.54_{\pm 2.51}$	$78.91_{\pm 1.99}$	$78.38_{\pm 3.24}$
	M^W_{TB}	Off	$43.48_{\pm 4.35}$	$70.60_{\pm 0.03}$	$78.60_{\pm 0.12}$
		Initial	$50.72_{\pm 2.51}$	$77.36_{\pm 2.17}$	$80.10_{\pm 2.65}$
		M^S_{TB}	$\mathbf{71.01}_{\pm 6.64}$	$\mathbf{82.58}_{\pm 0.34}$	$\mathbf{85.00}_{\pm 0.80}$

89.2%, respectively. These exceed the highest published reference scores shown for each benchmark, reported under their original protocols.

Harness choice affects same-model performance (Lewis, 2026). TORCH-GA started higher than Claude Code on Terminal-Bench (74.2% versus 61.8%), leaving less improvement headroom, and finished higher (85.4% versus 80.9%). Its off/on comparison retains evolved worker memory and measures added runtime supervision.

With Qwen and Terminus-2, TORCH increased success by 12.3 percentage points while reducing recorded input tokens by about 29%, including both roles. In contrast, Claude Code used more input and output tokens with supervision (Table 5).

Comparison with existing supervision systems. LivePlan also improved performance on both benchmarks using a GPT-5.3-Codex advisor. SupervisorAgent, with a Claude Sonnet 4.6 supervisor, improved on Terminal-Bench but showed essentially unchanged performance on SWE-bench. TORCH-supervised configurations attained higher final scores with Qwen3.8-27B-FP8 serving both roles. With Steer enabled, more tasks changed from success to failure than from failure to success, despite fewer worker calls. In one case, the critic misidentified the vector for the smallest singular value as that for the largest and directed the worker toward a slower solution (Appendix A.3). We next examine whether updating both the worker's and supervisor's memories improves performance over successive rounds compared with updating only one role's memory.

5.2 How Joint Evolution Improves Performance

Joint evolution achieves the highest mean later-round performance. Across three independent runs per condition, joint evolution averaged $64.33 \pm 4.01\%$ over rounds 6–10, versus $60.33 \pm 3.69\%$ for worker-only and $55.17 \pm 3.33\%$ for supervisor-only evolution. Round-10 means were $65.00 \pm 4.33\%$, $61.67 \pm 5.20\%$, and $58.33 \pm 7.64\%$, respectively (Figure 2(a)).

Joint evolution achieves higher mean failure recovery and success retention. Joint evolution averaged 31.4% failure recovery and 79.9% success retention, versus 29.9%/78.8% for worker-only and 26.1%/77.7% for supervisor-only (Figure 2(b)). From round 1 to 10, joint averaged 12.3 fail-to-pass and 4.7 pass-to-fail changes, versus 11.7/5.3 and 11.7/6.0 for the one-sided variants (c). These descriptive transitions reflect both memory changes and execution variability (Appendix D.4).

5.3 How Experience Transfers Across Models and Domains

Strong models benefit from accumulated experience. Supervisor-memory transfer added 6.8 points for GLM-5.3 and 2.3 for DeepSeek-V4.1-Flash with transferred worker memory, versus +1.1 and -1.1 with initial worker memory (Table 2). Using both transferred memories together yielded the highest scores: 88.8% for GLM-5.3 and 93.3% for DeepSeek-V4.1-Flash. Transferred supervisor memory provided larger gains when the worker also used transferred memory.

Experience transfers to another coding benchmark. FeatureBench tests reuse beyond the evolution benchmark within coding. Worker-memory transfer with supervision off raised mean pass rate from 13.04% to 43.48%. With transferred worker memory, supervisor-memory transfer added 20.29 points (50.72% to 71.01%); with initial worker memory, it added 1.45 points (26.09% to 27.54%).

Task differences help explain the transfer pattern. Despite its different target domain, Web shares editing and debugging operations with Terminal-Bench; Office requires different artifact processing. Worker-memory transfer without supervision improved Web's mean but not Office's. With initial supervisor memory, it added 4.39 points on Web versus 3.65 on Office; with initial worker memory, supervisor-memory transfer added 5.20 on Office versus 2.67 on Web (Table 3). This pattern is consistent with worker memory helping with familiar editing and debugging operations, while supervisor memory helps guide tasks that require different operations, such as Office artifact processing.

With transferred worker memory, supervisor memory added 5.22 points on Office and 4.90 on Web; using both transferred memories together yielded the highest mean scores. Appendix B.3 traces an OCR-derived rule redirecting an Office calendar task from reading toward output generation, reusing a lesson about when to begin delivery across different tasks.

5.4 Connecting Local Progress to Task Completion

Moving toward delivery also requires verifying task completion. In the webserver case (Appendix C), successful Git and shell operations coexisted with an empty HTTP response. Supervisor-guided diagnosis exposed a surviving process; the worker repaired the service and verified its content. The worker can learn to test the full deployment chain after a repair; the supervisor can learn to check the HTTP response even when Git and shell commands succeed.

In the video case, supervision redirected exploration toward a runnable script, but completion review missed inaccurate landing-frame predictions. HTTP content verified service recovery, whereas a runnable script did not establish accurate event detection. Supervisor learning must address both when to redirect execution and what evidence confirms completion.

6 Conclusion

TORCH combines online correction with separate worker and supervisor learning in TORCH-GA, a harness that fully implements the framework. Experiments show gains across harnesses, higher mean later-round performance from joint than one-sided evolution, and frozen-memory reuse across models and domains. Supervision need not increase total token consumption: Qwen–Terminus-2 improves performance with fewer recorded tokens across both roles.

Limitations. Even under identical settings, long-horizon tasks vary in execution paths, retries, and completion length, making actual token consumption and elapsed time difficult to match across conditions. Our resource measurements characterize observed usage; they do not establish performance at equal token consumption or execution time.

AI USE STATEMENT

Generative AI was used exclusively to assist with initial research ideation, literature retrieval, minor code implementation details, and manuscript language editing and polishing. It was not used for primary algorithm design, core experimental implementation, data collection, or empirical result verification. The authors reviewed all AI-assisted outputs and take full responsibility for the contents, code, and claims of this work.

REFERENCES

Anthropic. Claude Opus 4.8 System Card. System card, 2026. URL https://www.anthropic.com/claude-opus-4-8-system-card. Accessed September 18, 2026.

Yi Bin, Xiaoyang Yuan, Haoxi Zeng, Wencheng Ye, Wenqi Shao, Chen Qian, Wei Ye, Yujuan Ding, Zheng Wang, Pengpeng Zeng, et al. Terminal agents: A survey of AI agents in command-line environments. *arXiv preprint arXiv:2608.20485*, 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024.

Shubham Gandhi, Yiqing Xie, Atharva Naik, Ruichen Zhu, and Carolyn Rose. Steer, don't solve: Training small critic models for large code agents. *arXiv preprint arXiv:2606.21811*, 2026. URL https://arxiv.org/abs/2606.21811.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, volume 2024, pp. 57734–57811, 2024.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework, 2024. URL https://arxiv.org/abs/2308.00352.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, volume 2024, pp. 32808–32824, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Seth Karten, Joel Zhang, Tersoo Upaa Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in multi-turn conversation. In *International Conference on Learning Representations*, volume 2026, pp. 54738–54778, 2026.

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

Sydney Lewis. Same model, different harness: Different coding-agent results. *arXiv preprint arXiv:2608.26218*, 2026.

Jiaqing Liang, Jinyi Han, Weijia Li, Xinyi Wang, Zhoujia Zhang, Zishang Jiang, Ying Liao, Tingyun Li, Ying Huang, Hao Shen, et al. GenericAgent: A token-efficient self-evolving LLM agent via contextual information density maximization (V1.0). *arXiv preprint arXiv:2604.17091*, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

Fulin Lin, Shaowen Chen, Ruishan Fang, Hongwei Wang, and Tao Lin. Stop wasting your tokens: Towards efficient runtime multi-agent systems. In *International Conference on Learning Representations*, volume 2026, pp. 113497–113528, 2026a.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026b.

Jason Liu. When may an agent stop? evidence-carrying termination for tool-using LLMs. *arXiv preprint arXiv:2608.23623*, 2026.

Shuyang Liu, Saman Dehghan, Ji Young Kim, Jatin Ganhotra, Martin Hirzel, and Reyhaneh Jabbarvand. Online monitoring and corrective steering of programming agents. *arXiv preprint arXiv:2608.06701*, 2026.

Zhongwen Luan, Xiaoyu Zhang, Ming Hu, Yue Yang, Jiongchi Yu, and Xiaohong Chen. Repair or resample? rethinking failure debugging in LLM multi-agent systems. *arXiv preprint arXiv:2608.25920*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems, 2024. URL `https://arxiv.org/abs/2310.08560`.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023. URL `https://arxiv.org/abs/2304.03442`.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023.

Tencent WorkBuddy Bench Team, Siqi Cai, Shaopeng Chen, Xiang Fei, Yong Mao, Zihan Xu, Zhiheng Lyu, Zhijian Shao, Yuchen Shi, Shuwen Zhang, et al. Tencent WorkBuddy bench: A multi-domain coding-agent benchmark with contamination-resistant task construction. *arXiv preprint arXiv:2607.20911*, 2026.

Vals AI. SWE-bench Verified. `https://www.vals.ai/benchmarks/swebench`, 2026. Leaderboard updated September 1, 2026; accessed September 26, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023. URL `https://arxiv.org/abs/2305.16291`.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-Shepherd: Verify and reinforce LLMs step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, 2024a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents, 2024b. URL https://arxiv.org/abs/2402.01030.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yi Wang, Haopeng Zhang, Chengxiang Huang, Rui Dai, Kaikui Liu, Piotr Koniusz, and Xiangxiang Chu. LoopArena: Benchmarking models as runtime controllers for loop engineering. *arXiv preprint arXiv:2608.28281*, 2026.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation, 2023. URL https://arxiv.org/abs/2308.08155.

Yang Xiao, Yusong Sun, Haoyi Wu, Wenyang Hui, Wen Da, Zhaokai Luo, Mu Chuan, Yao Hu, Wenjie Li, and Chengyue Jiang. PILOT in the loop: Live self-improvement for long-horizon agents. *arXiv preprint arXiv:2608.26530*, 2026.

Yisen Xu, Chenglin Li, Zehao Wang, Jinqiu Yang, and Tse-Hsun Chen. Preventing premature commitment in coding agents with an evidence-conditioned execution layer. *arXiv preprint arXiv:2607.28815*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024.

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

Simon Yu, Derek Chong, Ananjan Nandi, Dilara Soylu, Jiuding Sun, Christopher D Manning, and Weiyan Shi. Shepherd: Enabling programmable meta-agents via reversible agentic execution traces. *arXiv preprint arXiv:2605.10913*, 2026.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning*, pp. 62138–62160. PMLR, 2024.

Qixing Zhou, JiaCheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=41xrZ3uGuI.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. Where LLM agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

A Inference and Supervision Details

We report inference settings and the TORCH-GA supervision interface; Figure 1 provides the TORCH framework overview.

A.1 Inference Configuration

Selected published references. Table 1(b) reports Vals AI evaluations on all 500 SWE-bench Verified tasks with mini-SWE-agent (Vals AI, 2026). Table 1 presents four reference model families shared across both benchmarks, with scores reported under their published evaluation protocols and inference budgets. In the Vals AI runs, GPT-5.5 uses xhigh reasoning effort and a 128k output-token limit. Claude Opus 4.8 and 4.7 use max effort, temperature 1, and a 128k limit. Gemini 3.1 Pro is the February 2026 Preview, with high reasoning effort, temperature 1, and a 65,536-token limit. The broader Vals AI leaderboard also includes GPT-5.6 Sol at 96.2%.

In TORCH experiments, each of the three models uses a 256k-token context window and matching worker/supervisor inference settings. Qwen3.8-27B-FP8 uses reasoning effort `low`; DeepSeek-V4.1-Flash and GLM-5.3 use their default reasoning and sampling settings. These settings remain fixed within each supervisor-off/on comparison.

Review settings. TORCH-GA schedules periodic reviews every 10 worker steps and triggers a stall review after 180 seconds without feedback, with additional reviews for follow-up, task-budget progress, and attempted completion. The standard correction control uses a threshold of three consecutive interventions, rather than limiting the total reviews per task. Review and correction settings are runtime controls and remain fixed within a matched off/on pair.

A.2 Native Supervision Baseline Configurations

SupervisorAgent uses Qwen3.8-27B-FP8 for both manager and worker, with `reasoning_effort=low`. Its supervisor, summary model, and verification sub-agent use Claude Sonnet 4.6 with thinking explicitly disabled (`thinking.type=disabled`).

Steer, Don't Solve uses the authors' modified mini-swe-agent 1.11.1: supervisor-off runs `DefaultAgent`, while supervisor-on runs `DefaultPRMAgent` with the authors' trained 4B critic. LivePlan uses SWE-agent 1.1.0 and SWE-ReX 1.4.0: supervisor-off disables the monitor/refiner, while supervisor-on enables the native rule monitor and advisor. Both systems use Qwen3.8-27B-FP8 as the worker. LivePlan uses GPT-5.3-Codex as the advisor because its original GPT-5.2-Codex was discontinued; Steer retains the authors' trained 4B critic. Each pair measures supervision within its own implementation.

A.3 Paired Outcomes and Feedback in Steer

We examine paired outcomes for all tasks in Table 1. Among 89 Terminal-Bench pairs, one task changed from failure with supervision off to success with supervision on, while three changed from success to failure. Among 500 SWE-bench Verified pairs, the corresponding counts were 23 and 31. On these paired sets, mean worker calls fell from 40.33 to 35.17 on Terminal-Bench (about 13%) and from 29.28 to 21.64 on SWE-bench (about 26%). Shorter execution did not translate into higher success; fewer calls can also reflect premature termination.

In `largest-eigenval`, feedback at step 15 incorrectly described `Vh[-1, :]` from `np.linalg.svd(...)` as the vector for the largest singular value and rejected its use for null-space estimation. NumPy orders singular values in descending order, so the last row is a right singular vector associated with the smallest singular value. The worker then pursued the suggested regularized solution, which ran substantially slower than the reference implementation. This task succeeded without supervision and failed with it. The trace establishes misleading feedback, but does not isolate that feedback as the sole cause of failure. Despite instructions to avoid prescribing specific technical solutions, observed feedback often specified algorithms, commands, and code locations. Of 593 Terminal-Bench feedback messages, 282 (47.6%) were labelled `Critical intervention required`.

A.4 Supervisor Observation and Access Boundaries

Table 4 summarizes the information available to the supervisor and its diagnostic access during each review. The default event snapshot excludes the worker's reasoning field; on-demand trajectory retrieval exposes the complete recorded event stream, including recorded reasoning and tool results.

Table 4: Supervisor interface implemented in TORCH-GA.

Snapshot exposed at each review	
Input	**Content**
Task	Original instruction
Review context	Review trigger and target
Current state	Most recent 20,000 characters
Observable events	Most recent 30 events
Supervisory history	Most recent 10 decisions
Workspace	Directory listing (up to two levels) and referenced-artifact status
Execution state	Turn usage and completion-review status
On-demand retrieval	Worker trajectory, loaded prompt, and supervisor memory
Diagnostic access and execution boundaries	
Resource or operation	**Access**
Trajectory, prompt, and supervisor memory	Retrievable for diagnosis
Terminal and process state	Inspectable
Diagnostic code	Bounded execution via `code_run`
Workspace and artifacts	Read-only under the supervisor's Unix identity
Supervisor scratch directory	Writable for auxiliary diagnostics
Final submission	Worker only
Hidden tests and grading data	Not exposed to the online task context
Host/verifier credentials	Access prohibited by the supervisor prompt

A.5 External Harness Adapters

External adapters and TORCH-GA both support inspecting execution and issuing corrections, but encode requests and decisions in different formats. External decisions include evidence, diagnosis, correction, confidence, state version, and target action ID; inspection requests read-only tools. TORCH-GA performs diagnostic calls before its final two-action response. Initial corrections state what the worker needs to satisfy or verify; if follow-up reviews show that the issue persists, later corrections may specify concrete steps. Implementations and prompt templates are in the code release.

A.6 Supervisor Prompts and Decision Contract

The following excerpts reproduce the static TORCH-GA system prompt and decision-prompt template verbatim, with paragraph breaks added for readability. Frozen supervisor memory and the current snapshot supply the task-specific context. Within each role-specific memory bank, L1 is a routing index, L2 stores stable facts across runs, and L3 stores reusable procedures supported by execution evidence. The runtime validates the response and assigns the intervention level and target. The `action` field specifies whether to inject a correction. The completion gate applies this same review process to attempted completion; it is a review trigger, not an additional decision type. Figure 3 summarizes the two actions.

System prompt: evidence and intervention (excerpts)

Judge observable progress, tool results, and artifacts.
When evidence is insufficient, inspect files, trajectory, or run a bounded check yourself before deciding whether to interrupt the Worker. Memory supplies inspection guidance, not proof that the current task is wrong; ground every intervention in current observable evidence.
Distinguish productive deep reasoning from action debt: long reasoning is acceptable after concrete tool evidence or a runnable artifact, but repeated promises to act, summaries, or plans without a tool call are not progress. Elapsed time alone is never drift when new observable evidence continues to arrive.
When uncertainty is genuine, request one reversible discriminative probe rather than a guessed final implementation. At completion, independently check every hard requirement, required file, and validation result; a Worker's completion claim alone is not sufficient evidence.

Decision prompt: output contract (excerpt)

First obtain any needed memory and public task evidence using the read-only tools. When ready to decide, return exactly one JSON object with keys action, reason, correction, level, target and no surrounding prose or Markdown. action is continue or intervene. For continue, correction must be empty, reason must be progressing and level=null. For intervene, correction must be one short and concrete next action grounded in the public task, trajectory, and L1/L2/L3 memory; it may name the required public path, command, or acceptance check, but must not solve the task or quote hidden grading data.

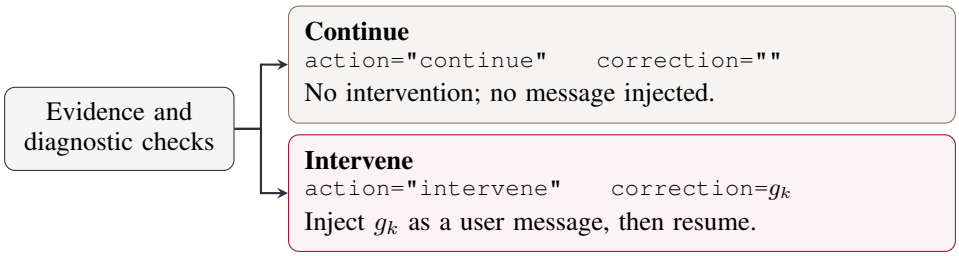

Figure 3: Supervisor decision contract implemented in TORCH-GA. The supervisor chooses whether to intervene: continue injects no message, while intervene injects corrective guidance. These actions apply to both execution-time reviews and reviews triggered by the completion gate. Execution and submission remain the worker's responsibility.

B MEMORY REFLECTION AND MERGING

Appendix A describes supervision during task execution. After evaluation, TORCH-GA uses the returned reward to generate role-specific candidate updates. Execution memories remain fixed within the round; candidates are merged separately at round end for the next round. This appendix gives the prompts and processing steps for generating and merging these updates.

B.1 REFLECTION AND CANDIDATE UPDATES

Once task evaluation finishes, the harness restores each role's saved conversation and supplies the same scalar task reward $r_{t,i}$ through the fixed prompt below. Hidden tests and grading criteria remain inaccessible during both execution and reflection. The role continues in the original task workspace and produces changes to its own memory as a candidate update, which is exported for round-end merging. In the template, {`reward`} is replaced with the observed reward.

Post-evaluation reflection prompt (full template)

Your reward for this trial is {reward}. Distill your trajectory and experience into your memory, then finish. The reward may be fractional partial credit: preserve its exact value, do not round it into binary success/failure, and do not invent failed criteria from the score alone.

B.2 MERGING AND SEMANTIC REVIEW

The update operator $\mathcal{U}_R$ in Section 3.5 processes candidates in deterministic execution order through three-way merging: it compares the round-start snapshot, the accumulated memory bank, and each candidate update. Every candidate-changed path then undergoes semantic review, including changes merged without textual conflicts. The reviewer receives the source versions, merge draft, and each candidate's reward and role-specific reflection.

The merge prompt is a fixed template instantiated with the role and changed paths. The following verbatim excerpt specifies how evidence should guide the review:

Memory merge prompt: evidence and consistency (excerpt)

Do not simply union all candidate text. Check duplication, semantic conflicts, opposite operational advice, unsupported causal claims, and consistency across files. Classify the evidence behind every proposed durable statement as an observed fact, a failure boundary, or an unexecuted hypothesis before editing.
A reward=1 candidate may contribute a successful recipe only when observable execution evidence supports it. A reward=0 candidate may contribute an observed fact, a failure boundary, or a conservative validation check. An unexecuted or unsuccessful suggestion must remain explicitly labelled as unverified and must never be described as correct, feasible, proven, causal, or reusable. When evidence cannot choose between opposing rules, rewrite them into one neutral verification principle or reject them; do not retain both opposing rules.

The reviewer edits the memory and records one `keep`, `merge`, `rewrite`, or `reject` decision per reviewed path, with a reason and evidence sources. Validation checks edit scope, decision coverage, memory structure, and unresolved conflict markers. If validation fails, the reviewer receives the error and revises the staged memory. Only validated updates become available to subsequent rounds. Updates do not alter the execution or evaluation result that generated a reflection.

B.3 SELECTED ROLE-SPECIFIC MEMORY EXCERPTS

We select service-deployment entries from the frozen source memory snapshot supplied to the cross-model evaluation. They illustrate how the two roles organize experience for different decisions. In the service-deployment examples below, worker excerpts are translated from Chinese and supervisor excerpts retain their original wording.

Worker: execute and verify the deployment chain. The L1 routing index points service tasks to `services_and_systems_sop.md`. Its L2 facts distinguish HTTP cache-wrapper errors from actual server responses. In L3, the deployment procedure states:

Worker memory: deployment verification (translated excerpt)

A reachable service does not establish that the push-to-deployment chain works. From a clean test state, perform clone → commit → push → HTTP retrieval of the newly pushed content, checking each step's exit status, service logs, and file permissions. An HTTP 404 may originate from an earlier clone or push failure.

The stored procedure specifies operations and checks that can be instantiated with the current task's repository, endpoint, and content.

Supervisor: distinguish repair progress from a need to intervene. The supervisor's L1 index routes Git-backed web deployment to `web_server_config_supervision_sop.md`. Its L3 rules include:

Supervisor memory: intervention and completion (excerpts)

Do not intervene for a single 403 that the worker is actively diagnosing and fixing.
HTTP response on the required port returns correct status code AND correct body content.
Do not require the worker to restart the service multiple times or perform redundant verification beyond what the task specifies.

These entries concern whether ongoing recovery warrants intervention, what evidence establishes completion, and when further guidance would be redundant. The two banks preserve complementary experience: a worker procedure for testing the deployment chain and supervisory rules for assessing progress and controlling intervention.

Observed reuse: from OCR extraction to calendar synchronization. Beyond stored excerpts, an Office calendar-synchronization trace shows the supervisor retrieving and applying a Terminal-Bench OCR lesson. This entry is absent from the initial SOP. The delivery-and-completion SOP copied into Office for transfer is byte-identical to the evolved source file, and the recorded memory-read response contains the entry.

Supervisor memory reused on an Office task

Stored OCR lesson (translated excerpt). "The correction must be a single concrete command and explicitly prohibit further extraction." The accompanying instruction requires writing the requested output from existing OCR data before any further OCR, calibration, or cropping.
Target-task observation. The worker continued reading and analyzing calendar data after two earlier requests to begin writing. The supervisor read the stored entry and explicitly referred to the "OCR case" when formulating its next correction.
Delivered correction (verbatim). "Stop reading. All source data is already on screen. Your next tool call must be a single code_run (type=python) that loads the three input JSON files, applies the sync logic, and writes all three output files. Do not call read_file or search_text again."

The worker's next call ran Python to update the task-state and synchronization-history JSON files and write the synchronization summary; the following call read back all three outputs for verification. The task scored 0.7871, passing 80 of 109 programmatic checks. This trace records OCR-derived guidance redirecting calendar processing toward delivery. Trace: `calendar-dida-sync-state__f5Rohnh`.

C Cases of Same-Model Trajectory Correction

The following cases illustrate runtime correction and its limits: repeated reviews guided recovery in the webserver task, while video processing shows how supervisory redirection can produce a runnable artifact whose prediction errors remain undetected at completion.

C.1 Webserver: Recovery Through Repeated Reviews

Task: Git push deploys `hello.html`; HTTP on port 8080 returns `hello world`. Both roles use GLM-5.3 with M_{Qwen} and separate contexts. Selected turns are summarized below; intervening steps are omitted.

Turns 1–5 Worker	Reads memory and inspects the environment. After a request to build at turn 4, creates the repository, deployment hook, and HTTP server. **Output:** `8080 OPEN`

· · · Turns 6–18: Git ownership and SSH setup · · ·

Turns 19–20 Worker	HTTP works. Resets the repository and web root for a fresh test, but leaves the old server running. **Output:** push succeeds; HTTP body is empty; shell exit code is 0.
After 20 Supervisor	**Flags the empty response.** Check the deployed file and hook, then rerun the clone–push–HTTP chain.
Turn 21 Worker	Checks the file, deployment hook, and HTTP response. **Output:** file exists; `hookrc=0`; `Empty reply from server.`
After 21 Supervisor	**Identifies the deleted working directory.** The server predates the web-root reset. Restart it and verify the HTTP response.
Turn 22 Worker	Attempts to restart the server. **Output:** `Address already in use.`
After 22 Supervisor	**Recognizes that the old server process is still alive.** Identify and terminate the actual server process; confirm the port is free before restarting.
Turns 23–24 Worker	Frees the port and restarts. HTTP recovers, but another web-root reset reproduces the failure.
Turn 25 Worker	Recognizes the repeated reset; removes the handler's dependency on the working directory with `directory='/srv/web'`. **Fresh clone–commit–push–HTTP test:** `hello world`.
Turn 26 Worker	**Recheck:** ports 22 and 8080 open; HTTP returns `hello world`.

The same-model supervisor identified the overlooked stale-process state; the worker implemented and verified the repair. Later completion-gate interventions were redundant and are omitted here. Post-run task evaluation returned reward = 1.

C.2 Video Processing: Redirection Without Accurate Completion

Task: Write `/app/jump_analyzer.py` to detect takeoff and landing frames in hurdle-jump videos and save them to `/app/output.toml`. Trace: `PREMkGe`, GLM-5.3 with supervision enabled. Selected turns are summarized; quoted excerpts reproduce recorded messages or outputs.

Turns 4–5 Worker	Probes frame differences and foreground-mask statistics instead of writing the required script.
After 4, 5 Supervisor	Requests script creation, execution, and verification of both output fields. **Excerpt after 5:** "No further probing before the artifact exists."
Turn 6 Worker	Writes `/app/jump_analyzer.py`. **Tool result:** `status: success`, 2,950 bytes written.
Turns 7–8 Worker	Runs a shell command through the Python tool, then retries with `type: bash`. **Outputs:** `SyntaxError: invalid syntax`, then exit code 0 and `takeoff 110 land 118`.
···	Turns 9–20: further motion analysis ···
After 20 Supervisor	**Excerpt:** "Stop probing and update /app/jump_analyzer.py with the frame-to-frame motion-diff approach"; also requests rerunning the example and checking the output fields.
Turns 21–22 Worker	Rewrites the script using frame-to-frame motion differences and reruns it. **Output:** `takeoff=54 land=61 -> /app/output.toml`; exit code 0.
Turn 23 Worker	Reports completion and describes the output as consistent with the observed motion data.
Completion Supervisor	**Recorded decision:** `action: continue`, `reason: progressing`; no correction.
Post-run Verifier	**Result:** 3 tests pass, 2 fail; reward = 0. Landing frame 61 falls outside [62, 64] on the example; landing frame 230 falls outside [231, 234] on the test video. These checks occur after execution.

The worker recovered from the early tool error and produced a runnable artifact after supervisory redirection. However, the final landing-frame predictions failed verification, despite the supervisor's completion-time `continue` decision. This case separates progress toward a deliverable from accurate task completion.

D RESOURCE USAGE

We report aggregate token usage for supervisor-off/on comparisons and task-level resource measurements during TORCH-GA memory evolution.

D.1 TOKEN USAGE WITH RUNTIME SUPERVISION

Table 5 reports aggregate token usage on Terminal-Bench 2.1 for Qwen3.8-27B-FP8 with Terminus-2, Claude Code, and TORCH-GA, and for GLM-5.3 and DeepSeek-V4.1-Flash with TORCH-GA. Qwen configurations follow Table 1; GLM and DeepSeek follow Table 2. Each off/on pair fixes the worker model, harness, and worker memory. On totals include both roles; Off totals include only the worker.

The higher task scores in Table 1 and Table 2 accompany different resource changes across configurations (Table 5). With Qwen3.8-27B-FP8 and Terminus-2, total input decreases from 813.4M to approximately 577.8M tokens. Its recorded uncached and cached components decrease from 551.1M to 393.3M and from 262.3M to 184.4M, respectively, while output increases from 18.7M to 20.6M. With Claude Code, uncached input, cached input, and output all increase. With TORCH-GA, uncached input, cached input, and output decrease from 176.2M to 128.7M, 433.7M to 333.4M, and 19.9M to 13.3M. For GLM-5.3 with TORCH-GA, output decreases from 1.3M to 1.0M tokens while both input categories increase; for DeepSeek-V4.1-Flash, all three categories increase.

Table 5: Aggregate recorded token usage on Terminal-Bench 2.1 (millions). On includes both worker and supervisor usage; Off includes worker usage only. Qwen configurations follow Table 1, with TORCH-GA using M_{19}. GLM and DeepSeek use transferred $M_{\text{Qwen}} = M_{20}$, as in Table 2. Input tokens are split into uncached and cached tokens.

		Uncached Input (M)		Cached Input (M)		Output (M)	
Model	Harness	Off	On	Off	On	Off	On
Qwen3.8-27B-FP8	Terminus-2	551.1	393.3	262.3	184.4	18.7	20.6
	Claude Code	244.4	268.1	116.4	125.7	9.3	23.3
	TORCH-GA	176.2	128.7	433.7	333.4	19.9	13.3
GLM-5.3	TORCH-GA	6.4	14.8	32.7	38.0	1.3	1.0
DeepSeek-V4.1-Flash	TORCH-GA	7.0	16.1	66.0	82.1	3.3	3.7

D.2 TOKEN USAGE AND INTERACTION COUNTS

We examine how execution resource usage changes during the first joint-evolution run in TORCH-GA. Figure 4 and Table 6 report measurements for Qwen3.8-27B-FP8 on the 40-task continuation study in Section 4.3, with one execution per task in each of local rounds 1–10, starting from $\widetilde{M}_0 = M_5$. The figure shows means and medians across tasks; the table provides means and additionally reports supervisor calls.

Resource usage changes differently across metrics. From round 1 to round 10, mean input tokens increase from 1.446M to 1.886M per task (30.4%), while mean output tokens decrease from 75.3k to 68.0k (9.7%). Mean worker turns increase from 43.0 to 47.9 (11.4%). Output generation decreased, while input processing and worker turns increased.

These changes are non-monotonic: mean input tokens, output tokens, and worker turns all peak in round 7, at 2.073M, 88.0k, and 53.8, respectively. Mean supervisor calls range from 7.3 to 8.7 across rounds and end at 7.5. The measurements therefore distinguish changes in token processing, text generation, and interaction length during memory evolution.

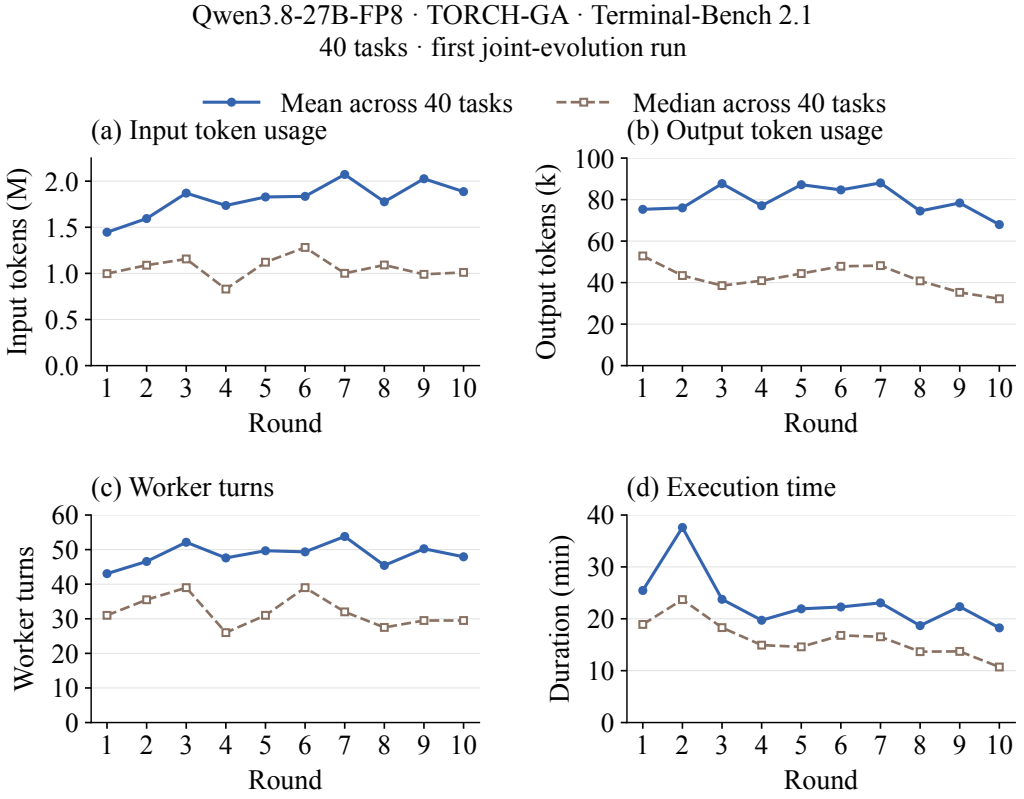

Figure 4: Resource usage during memory evolution in TORCH-GA, with means consistently above medians. Each local round contains one execution of each task in the fixed 40-task set in the first joint-evolution run. Input and output tokens include both worker and supervisor; M/k denote million/thousand tokens. Curves show task-level means and medians within each round.

D.3 Variation Across Tasks and Recorded Duration

Across all ten rounds, the mean exceeds the median for each of the four metrics in Figure 4. This consistent gap indicates that higher-consumption executions pull the averages above the median task. Reporting both statistics therefore reveals variation across tasks that the means in Table 6 alone would conceal.

Recorded duration in the first joint-evolution run provides a further view of execution resource usage: its mean decreases from 1,528 seconds in round 1 to 1,095 seconds in round 10, a 28.3% reduction, with substantial intermediate variation and a peak of 2,256 seconds in round 2. Duration covers initialization, execution, supervision, and finalization, excluding environment provisioning, subsequent verification, and cross-round memory updates. Recorded durations include interrupted and resumed executions; timing conditions varied across runs.

Table 6: Mean resource usage per task in each local round of the first joint-evolution run in TORCH-GA on the 40-task set. Supervisor calls supplement the four metrics in Figure 4. Input and output tokens include both roles; M/k denote million/thousand tokens. Duration is reported in seconds here and minutes in the figure.

Round	Input (M)	Output (k)	Worker turns	Sup. calls	Duration (s)
1	1.446	75.3	43.0	7.3	1,528
2	1.595	76.0	46.6	8.1	2,256
3	1.870	87.7	52.1	8.5	1,425
4	1.736	77.1	47.6	8.2	1,183
5	1.828	87.2	49.7	8.2	1,315
6	1.835	84.7	49.4	8.0	1,337
7	2.073	88.0	53.8	8.7	1,384
8	1.776	74.5	45.4	7.6	1,121
9	2.026	78.4	50.2	8.4	1,340
10	1.886	68.0	47.9	7.5	1,095

D.4 Task Transitions Across Evolution Runs

For each run, let $n_{FP}, n_{FF}, n_{PP}, n_{PF}$ count failure-to-pass, failure-to-failure, pass-to-pass, and pass-to-failure transitions over all 40 tasks and nine adjacent-round pairs. Failure recovery is $n_{FP}/(n_{FP} + n_{FF})$; success retention is $n_{PP}/(n_{PP} + n_{PF})$. Endpoint counts compare each task once between rounds 1 and 10. Each independent run contributes one value to the reported mean and sample standard deviation across three runs.

Joint evolution has recovery $31.43 \pm 6.36\%$ and retention $79.87 \pm 4.99\%$; worker-only has $29.93 \pm 7.35\%$ and $78.83 \pm 6.73\%$; supervisor-only has $26.09 \pm 4.74\%$ and $77.69 \pm 5.92\%$. Endpoint fail-to-pass and pass-to-fail counts are 12.33 ± 2.52 and 4.67 ± 1.53 for joint, 11.67 ± 4.04 and 5.33 ± 1.53 for worker-only, and 11.67 ± 4.04 and 6.00 ± 2.00 for supervisor-only.

E FeatureBench-lite Evaluation Subset

Table 3 evaluates the following 23 tasks from FeatureBench-lite that require no GPU. The subset contains 21 `lv1` tasks and two `lv2` tasks; these suffixes denote difficulty levels, not dataset versions. The exact task identifiers are listed below.

```
astropy__astropy.b0db0daa.test_quantity_erfa_ufuncs.3fc54395.lv1
astropy__astropy.b0db0daa.test_table.48eef659.lv1
lightning-ai__pytorch-lightning.126fa6f1.test_connector.825ef3b1.lv1
lightning-ai__pytorch-lightning.126fa6f1.test_data.c8b292af.lv1
mlflow__mlflow.93dab383.test_databricks_tracing_utils.8ef44eb4.lv1
mlflow__mlflow.93dab383.test_judge_tool_search_traces.cad8fc4b.lv1
mlflow__mlflow.93dab383.test_responses_agent.072c4133.lv1
mlflow__mlflow.93dab383.test_serialization.2c029be6.lv1
mlflow__mlflow.93dab383.test_span.69efd376.lv1
mlflow__mlflow.93dab383.test_trace.17fde8b0.lv1
mlflow__mlflow.93dab383.test_validation.2e1c5076.lv1
mwaskom__seaborn.7001ebe7.test_algorithms.1f0181c2.lv1
mwaskom__seaborn.7001ebe7.test_regression.ce8c62e2.lv1
mwaskom__seaborn.7001ebe7.test_statistics.0f2ae277.lv1
netflix__metaflow.b390a8d4.test_stub_generator.7bf08c98.lv1
pandas-dev__pandas.82fa2715.test_concat.ebe5de39.lv1
pandas-dev__pandas.82fa2715.test_iceberg.85771c70.lv2
pandas-dev__pandas.82fa2715.test_list_accessor.7ab0b2ea.lv1
pydantic__pydantic.e1dcaf9e.test_deprecated_fields.40a2ec54.lv1
pydata__xarray.97f3a746.test_backends_chunks.fa55f68a.lv1
pydata__xarray.97f3a746.test_treenode.aa8ba777.lv2
sphinx-doc__sphinx.e347e59c.test_domain_c.4068b9e8.lv1
sympy__sympy.c1097516.test_nullspace.f14fc970.lv1
```