CONTINUAL POLICY PROGRAMMING FOR LLM AGENTS: LEARNING EXECUTABLE POLICIES FROM ONLINE EXPERIENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents can solve increasingly complex long-horizon tasks, while continually improving from ongoing interaction remains an important challenge. We introduce *Continual Policy Programming*, a framework that enables a frozen LLM agent to learn from fine-grained online experience by updating a persistent executable policy. Policy updates take effect during the current task and persist across future tasks, allowing newly acquired knowledge to accumulate over time. Across ALFWorld, WebShop, MiniGrid, and AppWorld, Continual Policy Programming enables agents to recover from failures online, reuse newly acquired behaviors across subsequent tasks, and accumulate capabilities over time. Our results show that persistent executable policies provide an effective way to turn streaming interaction experience into cumulative behavioral improvement.

1 INTRODUCTION

Large Language Model(LLM) agents have become increasingly capable of handling long-horizon, interactive tasks by adapting their behavior to evolving observations and context (Sun et al., 2023; Yao et al., 2022b). As agents operate over interactions, they encounter new tasks, states, and failures that provide valuable experience for improving future behavior (Hu et al., 2026a; Majumder et al., 2023; Shinn et al., 2023; Wang et al., 2024b; Zhao et al., 2024). Therefore, enabling agents to accumulate such experience into reusable capabilities is an important step toward agents that can improve over time (Shinn et al., 2023; Wang et al., 2023; Zheng et al., 2025a). Existing work has explored this direction from several perspectives, including updating model parameters (Chen et al., 2023; Zeng et al., 2024), retaining experience through memory (Hu et al., 2026a; Park et al., 2023; Wang et al., 2024b; Zhong et al., 2024), and acquiring reusable skills that can guide subsequent behavior (Chen et al., 2026; Wang et al., 2023; Yang et al., 2026; Zheng et al., 2025a).

Learning from ongoing interaction requires improvements that are both *timely* and *persistent*. New experience can become relevant within only a few subsequent interactions, making it important for an agent to benefit from what it has just learned without waiting for an episode or a large collection of trajectories to finish. At the same time, useful improvements should persist beyond the current task and contribute to the agent's future competence. Existing adaptation mechanisms face different trade-offs in satisfying these two requirements. Parameter updates can internalize experience into behavior, but are costly and potentially unstable when performed frequently from only a few interactions (Du & Nguyen, 2023; Fawi, 2024; Shi et al., 2025; Zheng et al., 2025b). Memory can capture new experience immediately, but its influence on future behavior depends on how stored experience is organized, retrieved, and interpreted at subsequent decisions (Hu et al., 2026a;b; Wei et al., 2025; Xu et al., 2026). Reusable skills provide persistent procedural knowledge, but forming a generalizable skill typically requires sufficient evidence to identify recurring patterns, making them less suited to incorporating fine-grained experience as it arrives (Wang et al., 2026a;b; Yang et al., 2026; Yu et al., 2026; Zheng et al., 2025a). This leaves open how an agent can continually turn streaming experience into behavioral improvements that take effect quickly and accumulate reliably over time.

We address this challenge through **Continual Policy Programming(CPP)**, in which learning is interleaved with agent interaction and accumulated into a persistent executable policy represented as

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.

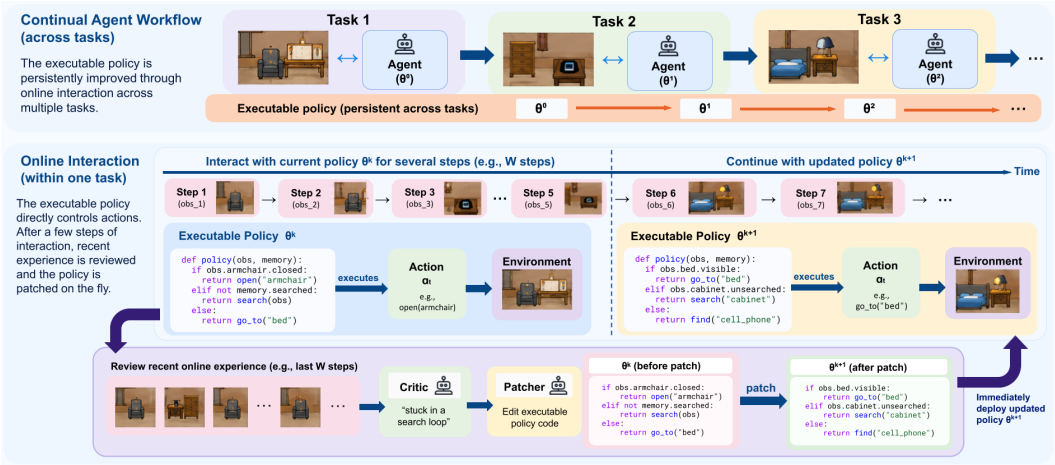

Figure 1: Overview of **Continual Policy Programming (CPP)**. An executable policy directly selects actions from observations and working memory during online interaction. After a short interaction window, a critic reviews recent experience and a patcher edits the policy code. The updated policy immediately controls subsequent actions and persists across tasks, allowing the agent to improve through continued interaction.

code. The policy directly controls the agent's actions and manages a task-specific working memory that records information needed during the current task. After a short interaction window, a critic examines the recent experience and determines whether it reveals useful knowledge that should change the agent's behavior. When an update is needed, a patcher incorporates this by editing the executable policy, including its action logic and the way working-memory variables are created, updated, and used. The revised policy is executed immediately in subsequent interactions and retained across task boundaries, while working memory is initialized for each task. Through repeated interaction, evaluation, and patching, knowledge discovered from immediate experience can improve behavior within the current task and accumulate into an executable policy that guides future tasks.

This learning process gives the agent three important capabilities. First, learning at the timescale of ongoing interaction enables **online failure recovery**. When the agent makes a mistake or reaches a state that its current policy does not handle correctly, the resulting experience can trigger an update that changes how the agent behaves in the remainder of the task and pushes the agent back towards success. Second, because these updates are retained in the persistent policy, they can **generalize** beyond the interaction that produced them. A newly observed discrepancy may reveal a missing condition, action, or procedure in the current policy, and incorporating it can improve behavior on future unseen tasks that require the same knowledge. Over time, the policy can therefore expand from handling previously encountered situations to covering new behaviors and task requirements. Third, these improvements can be incorporated with **little regression** to existing capabilities. Updates target the part of the policy implicated by the new experience, while the executable policy allows the agent to check whether the new behavior works as intended and whether previously successful behavior is preserved. Together, timely updates allow the agent to recover from failures as they occur, while persistent and controlled updates turn these local experiences into broader capabilities that accumulate over time.

We evaluate our approach on ALFWorld (Shridhar et al., 2020), WebShop (Yao et al., 2022a), MiniGrid (Chevalier-Boisvert et al., 2023), and Appworld (Trivedi et al., 2024), covering environments with different task structures and interaction dynamics. Across these benchmarks, Continual Policy Programming consistently improves the agent as interaction proceeds. The learned updates can recover trajectories that would otherwise fail, transfer newly acquired behavior to unseen tasks, and preserve previously successful behaviors as the policy continues to evolve. We further find that these gains emerge from fine-grained online learning, where useful knowledge discovered from only a few interactions can already improve subsequent decisions and remain available across future tasks. These results show that streaming interaction experience can be progressively incorporated into an executable policy, allowing an agent to build broader and more reliable capabilities over time.

Our contributions are threefold.

- We introduce **Continual Policy Programming(CPP)**, a framework for learning from fine-grained online interaction by continually updating a persistent executable policy. Policy updates can take effect within the current task and persist across task boundaries, allowing experience acquired during interaction to directly shape future behavior.
- We formulate the persistent executable policy as a unified representation of both decision-making and information management. The policy determines how the agent acts, what task-specific information is maintained in working memory, and how that information is used throughout interaction. This design allows online experience to be incorporated as reusable behavioral knowledge in the persistent policy, while transient task state remains local to the current task.
- We evaluate Continual Policy Programming on ALFWorld, WebShop, MiniGrid, and AppWorld, and show that it enables agents to recover from failures during ongoing tasks, reuse newly acquired behaviors across subsequent tasks and accumulate broader capabilities over the task stream. Our results show that fine-grained online experience can accumulate into broader and more reliable competence over time.

2 Related Works

Learning from interaction experience. LLM agents can improve over time by retaining and reusing information from previous interactions. Early approaches such as Reflexion Shinn et al. (2023), ExpeL Zhao et al. (2024), and CLIN Majumder et al. (2023) preserve experience as textual feedback or memory that informs future decisions. Later work moves toward more structured and reusable knowledge, including workflows Wang et al. (2024b) and executable skills Wang et al. (2023); Zheng et al. (2025a). These methods establish interaction experience as an important source of persistent knowledge. We study a finer-grained setting in which knowledge from only a few recent interactions is incorporated into a persistent policy while interaction is still ongoing.

Executable programs for agent behavior. Executable programs provide a structured way to represent agent behavior. Code as Policies Liang et al. (2023) demonstrates that LLMs can synthesize reusable policy programs, while CodeAct Wang et al. (2024a), AdaPlanner Sun et al. (2023), and ReCode Yu et al. (2025) use code to support interactive planning and adaptation during execution. More recent work further studies how executable policies can be improved from experience. IPP Aravindan et al. (2025) improves reusable policy programs through repeated evaluation over a training set followed by feedback-driven refinement. Kintsugi Cao et al. (2026) performs rollout-intensive repair of executable policies, using hundreds of evaluation rollouts to validate and select updates before freezing the resulting policy for test-time deployment. Our work studies a different setting of *test-time continual policy learning*. A few interactions from the ongoing task can trigger a persistent policy update, the revised policy immediately guides subsequent actions in the same task, and learning continues as the agent encounters future tasks.

Self-evolving agent systems and harness optimization. A parallel line of work treats the program surrounding the model as the object of optimization. Meta-Harness Lee et al. (2026) searches over harness code with an agentic proposer that reads the source, scores, and traces of prior candidates. Continual Harness Karten et al. (2026) moves adaptation inside a single run, editing prompts, sub-agents, skills, and memory. Related efforts refine harness code from environment feedback in a given domain Lou et al. (2026) or evolve and route among a family of harnesses over open-ended streams Liu et al. (2026). Our setting differs in what is being learned and how learning occurs. We study a single executable policy that is itself the decision procedure, and ask how experience arriving during interaction can be written into it.

3 Method

3.1 Preliminaries

Environment. We consider a sequence of episodic partially observable tasks. For episode e, let $g_e \in \mathcal{G}$ denote the task specification, such as a natural-language instruction, and let H_e denote the maximum number of interaction steps allowed for the task. The environment is modeled as an

episodic partially observable Markov decision process

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T, \Omega, \mu_0), \tag{1}$$

where $\mathcal{S}$ is the latent state space, $\mathcal{A}$ is the action space, $\mathcal{O}$ is the observation space, $T(s' \mid s, a)$ is the transition function, $\Omega(o \mid s)$ is the observation function, and μ_0 is the initial-state distribution. At the beginning of episode e, the environment samples an initial latent state $s_{e,0} \sim \mu_0$ and gives an initial observation $o_{e,0} \sim \Omega(\cdot \mid s_{e,0})$ to the agent.

Interaction protocol. At interaction step t, the agent conditions on the task specification g_e and its interaction history

$$h_{e,t} = (o_{e,0}, a_{e,0}, o_{e,1}, \ldots, a_{e,t-1}, o_{e,t}), \tag{2}$$

and selects an action $a_{e,t} \in \mathcal{A}$. The environment then transitions according to

$$s_{e,t+1} \sim T(\cdot \mid s_{e,t}, a_{e,t}), \tag{3}$$

and generates the next observation $o_{e,t+1} \sim \Omega(\cdot \mid s_{e,t+1})$ together with a termination indicator $d_{e,t} \in \{0, 1\}$. Some environments additionally provide an optional scalar reward $r_{e,t} \in \mathbb{R}$ to the agent, representing current progress towards task success. The episode terminates when the environment signals termination or the maximum interaction horizon is reached, formally,

$$T_e = \min\{t + 1 \mid d_{e,t} = 1 \text{ or } t + 1 = H_e\}. \tag{4}$$

The resulting interaction trajectory is

$$\tau_e = (g_e, o_{e,0}, a_{e,0}, o_{e,1}, a_{e,1}, \ldots, o_{e,T_e}), \tag{5}$$

with optional rewards $\{r_{e,t}\}_{t=0}^{T_e-1}$ included when they are provided by the environment.

Agent and evaluation. Writing $\mathcal{H}$ for the space of interaction histories, an agent is a map $\pi : \mathcal{G} \times \mathcal{H} \to \mathcal{A}$. Upon completion of each episode, the environment returns a task-success indicator $y_e \in \{0, 1\}$ to the agent, where $y_e = 1$ indicates successful completion of g_e. We consider a stream of N tasks,

$$\mathcal{D} = (g_1, g_2, \ldots, g_N), \tag{6}$$

and use task success rate as the primary evaluation metric $\mathrm{SuccessRate} = \frac{1}{N}\sum_{e=1}^{N} y_e$.

3.2 The Executable Policy

Starting from this section, we introduce our method. We first describe the executable policy, the program that produces the agent's actions, and how it interacts with the environment. Section 3.3 then describes how it is written and revised.

An *executable policy* is a triple

$$\theta = (\mathcal{W}, \mathsf{U}, \mathsf{P}), \tag{7}$$

where $\mathcal{W}$ is a working-memory schema, the *memory operator* $\mathsf{U} : \mathcal{W} \times \mathcal{O} \to \mathcal{W}$ maintains the memory, and the *action operator* $\mathsf{P} : \mathcal{W} \to \mathcal{A} \times \mathcal{W}$ uses it.

The schema declares the fields the agent maintains. At the start of an episode the agent instantiates it, producing a working memory $m_0 \in \mathcal{W}$ with its fields empty, which is filled and revised as the task proceeds and discarded when it ends.

The agent runs θ in the environment. Each step t has two phases,

$$\tilde{m}_t = \mathsf{U}(m_t, o_{e,t}), \qquad (a_{e,t}, m_{t+1}) = \mathsf{P}(\tilde{m}_t). \tag{8}$$

Concretely, upon receiving the raw observation $o_{e.t}$, the two phases divide the work of handling the observation and generating the action. U sees the raw observation and decides what should be fed into the memory. P never sees the raw observation and works from structured memory. In other words, the agent acts using P based on what U chose to record. The agent then executes $a_{e,t}$, receives $o_{e,t+1}$, and repeats until the episode terminates. Notably, the action comes from running code and does not directly rely on LLM calls.

Algorithm 1 Continual Policy Programming (CPP)

Require: task stream $\mathcal{D} = (g_1, \ldots, g_N)$; exemplars $\mathcal{F}_{\mathrm{dec}}$; review interval W

$\theta_0 \leftarrow \varnothing$
for $e = 1, \ldots, N$ **do** ▷ task scale
  $z_e \leftarrow \mathsf{Dec}(g_e; \mathcal{F}_{\mathrm{dec}})$
  $\theta_e^{(0)} \leftarrow \mathsf{Pat}(g_e, z_e, \theta_{e-1})$ ▷ initialize from the previous policy
  $m \leftarrow$ empty memory of schema $\mathcal{W}(\theta_e^{(0)})$
  observe $o_{e,0}$; $t \leftarrow 0$; $k \leftarrow 0$
  while episode not terminated **do** ▷ review scale
    $\rho_e^{(k)} \leftarrow [o_{e,t}]$; $(\mathsf{U}, \mathsf{P}) \leftarrow \theta_e^{(k)}$
    for $i = 1, \ldots, W$ **do** ▷ step scale
      $\tilde{m} \leftarrow \mathsf{U}(m, o_{e,t})$
      $(a_{e,t}, m) \leftarrow \mathsf{P}(\tilde{m})$
      execute $a_{e,t}$; observe $o_{e,t+1}$
      append $(a_{e,t}, o_{e,t+1})$ to $\rho_e^{(k)}$; $t \leftarrow t + 1$
      if episode terminated **then**
        break
      end if
    end for
    if episode terminated **then**
      break ▷ no review after a terminal window
    end if
    $(c_e^{(k)}, \delta_e^{(k)}) \leftarrow \mathsf{Cri}(g_e, \rho_e^{(k)}, \theta_e^{(k)})$
    if $c_e^{(k)} = 1$ **then**
      $(\theta_e^{(k+1)}, \mu) \leftarrow \mathsf{Pat}(g_e, z_e, \rho_e^{(k)}, \delta_e^{(k)}, \theta_e^{(k)})$
    else
      $\theta_e^{(k+1)} \leftarrow \theta_e^{(k)}$; $\mu \leftarrow \mathrm{id}$
    end if
    $m \leftarrow \mu(m)$; $k \leftarrow k + 1$ ▷ environment state and t are unchanged
  end while
  receive y_e; $\theta_e \leftarrow \theta_e^{(k)}$ ▷ discard task-local memory and trajectory
end for
return $\mathrm{SuccessRate} = \frac{1}{N} \sum_{e=1}^{N} y_e$

3.3 Online Patching: Updating the Executable Policy

In this section, we introduce how we update our executable policy θ. Notably, two things about θ make it a workable target for online updates. First, a failure can be traced to one piece of code. A bad action came either from U leaving out something g_e depends on, or from P mishandling what was recorded. Each is a named routine in the source, so a fix can be written for that routine while the rest of the policy stands. Second, a policy can be handed from one task to the next without dragging stale facts along. θ holds a way of keeping and using information, while a working memory holds what one episode actually turned up. Where a particular key was found therefore stays out of θ, and the routine that looks for keys and records where they are stays in.

As an overview, two LLM operators act on θ, a patcher Pat that writes code and a critic Cri that reads recent interaction and decides when code should change. Algorithm 1 summarizes the above procedure. We now follow one task from start to finish to see how these two operators work.

Entering a task. Task e opens with a decomposition of its goal,

$$z_e = \mathsf{Dec}(g_e; \mathcal{F}_{\mathrm{dec}}), \tag{9}$$

an ordered list of subgoals obtained from a small set of few-shot exemplars $\mathcal{F}_{\mathrm{dec}}$, each subgoal paired with the state variables it depends on. The variables suggest fields of $\mathcal{W}$ and the subgoals suggest branches of P, so one decomposition sketches both parts of the policy. We do not ask z_e to be correct, since it narrows the space the patcher searches over and a misreading of the goal can be worked around in the code once the critic points it out.

The patcher then writes the policy the task starts under, beginning from the one the previous task ended with,

$$\theta_e^{(0)} = \mathsf{Pat}(g_e,\, z_e,\, \theta_{e-1})\,, \qquad \theta_0 = \varnothing, \tag{10}$$

Running and reviewing. Let k index policy versions within a task. The agent runs $\theta_e^{(k)}$ for up to W action-observation pairs, recorded as $\rho_e^{(k)}$; working memory updates at each step while the code remains fixed. Before the next step is taken, the critic generates

$$\Big(c_e^{(k)},\, \delta_e^{(k)}\Big) = \mathsf{Cri}\Big(g_e,\, \rho_e^{(k)},\, \theta_e^{(k)}\Big)\,, \qquad c_e^{(k)} \in \{0,1\}, \tag{11}$$

where $c_e^{(k)}$ decides whether to revise and $\delta_e^{(k)}$ is a list of revision points in natural language. The critic reads the segment three ways. First, it asks whether the actions taken are carrying the agent towards g_e, or have it stuck, repeating itself, or working on something the goal does not require. Second, it asks whether the observations match what $\theta_e^{(k)}$ expected when it chose those actions, since a gap there means the code is holding a belief about the environment that the environment has just contradicted. Third, it asks whether the subgoals and state variables still represent g_e correctly, since a mistaken decomposition will keep producing sound-looking code that solves the wrong problem.

For the points raised, the critic names where in the policy the change should apply, either a memory field together with its update rule in U or a branch of P, and says what to change. On the other hand, the critic is also aware of what parts of the policy function as expected according to the goal, policy code and observation-action segment, so it can indicate what parts in the code should remain unchanged. Then, the patcher rewrites based on critic's feedback:

$$\Big(\theta_e^{(k+1)},\, \mu_e^{(k)}\Big) = \begin{cases} \mathsf{Pat}\Big(g_e,\, z_e,\, \rho_e^{(k)},\, \delta_e^{(k)},\, \theta_e^{(k)}\Big)\,, & c_e^{(k)} = 1, \\ \Big(\theta_e^{(k)},\, \mathrm{id}\Big)\,, & c_e^{(k)} = 0, \end{cases} \tag{12}$$

where $\mu_e^{(k)} : \mathcal{W}\big(\theta_e^{(k)}\big) \to \mathcal{W}\big(\theta_e^{(k+1)}\big)$ carries the working memory onto the schema of the new policy. Namely, the patcher will indicate for each field of the new schema whether to retain its value, initialize it anew, or derive it from existing memory. The patcher decides this because it has just changed how those fields are written and used, and knows which of the agent's existing records are still valid and meaningful under the updated policy. The agent then continues from its current environment state.

The interval W sets how finely the agent learns. A small W catches a problem quickly, but it also costs more critic calls, and many of them could indicate no change is required. On the other hand, a large W spends fewer calls. However, it lets a faulty policy run longer, and the states the agent reaches can be more difficult to recover from. In our empirical study in section 4, we set $W = 5$.

Leaving a task. Writing K_e for the number of reviews before termination, the task ends under $\theta_e^{(K_e)}$, and this is the policy θ_e that task $e+1$ begins from. Everything else the agent built in the task is dropped, including the working memory since its contents describe an episode that is over, and the decomposition z_e since it was a reading of previous goal.

3.4 Properties

The mechanisms above give the three capabilities:

Online failure recovery. At the next review after a failure, the critic reads the observations and actions that led there and diagnoses the cause. The patcher applies the corresponding fix to the code, and the agent continues from its current state under the revised policy. This closes the loop between failure and correction within a single episode, allowing recovery from mistakes as they happen.

Generalization. Revisions on the policy accumulate and cross the task boundary, and the specific instance of the working memory is dropped after a task is done. Hence, the reusable procedures are carried out through different tasks and the task-specific information is eliminated, enabling generalization across different tasks.

Little regression. The critic works from the policy and the segment, so it can say what to change and what to keep, and the patcher rewrites under both. Behavior that earlier tasks put in place is therefore left alone unless the current failure gives a reason to revisit it, and a policy that has accumulated capabilities over a long stream is not rewritten wholesale each time something goes wrong.

Table 1: Results over three seeds (mean ± standard deviation). ALFWorld, WebShop SR, MiniGrid SR, and AppWorld SR report task success rate (%); WebShop Reward retains its original benchmark scale. Bold values indicate the highest mean within each backbone and metric.

Backbone	Method	ALFWorld	WebShop		MiniGrid	AppWorld
		Overall SR	SR	Reward	SR	SR
GPT-4.1	ReAct	80.6 ± 2.6	33.0 ± 2.7	36.7 ± 2.0	41.3 ± 5.1	41.0 ± 4.4
	Reflexion	83.6 ± 2.7	39.7 ± 6.7	41.6 ± 4.5	45.7 ± 3.1	48.3 ± 3.2
	ReCode	84.6 ± 2.4	50.3 ± 1.2	61.9 ± 2.1	47.3 ± 2.1	64.7 ± 10.0
	ReMem	83.1 ± 3.8	54.0 ± 3.6	73.5 ± 3.5	56.0 ± 3.0	62.3 ± 3.5
	SkillAdaptor	81.1 ± 0.4	46.3 ± 3.2	57.9 ± 4.9	67.7 ± 7.5	67.0 ± 10.8
	CPP (Ours)	$\mathbf{98.5 \pm 1.3}$	$\mathbf{65.7 \pm 3.5}$	$\mathbf{79.4 \pm 2.6}$	$\mathbf{94.3 \pm 2.1}$	$\mathbf{76.7 \pm 2.5}$
DeepSeek-V4-Flash	ReAct	83.8 ± 1.7	22.0 ± 2.7	24.7 ± 2.2	53.7 ± 2.5	52.0 ± 5.6
	Reflexion	83.6 ± 4.9	26.7 ± 2.1	28.4 ± 3.6	53.3 ± 3.5	49.3 ± 6.1
	ReCode	90.8 ± 2.6	25.0 ± 7.0	31.0 ± 6.8	50.7 ± 4.7	60.0 ± 4.6
	ReMem	75.4 ± 4.7	36.3 ± 1.5	45.8 ± 1.1	64.7 ± 2.5	64.3 ± 7.0
	SkillAdaptor	85.6 ± 1.7	39.7 ± 7.0	48.8 ± 8.3	57.3 ± 10.7	70.7 ± 1.5
	CPP (Ours)	$\mathbf{99.0 \pm 1.1}$	$\mathbf{66.3 \pm 2.5}$	$\mathbf{78.1 \pm 2.1}$	$\mathbf{96.7 \pm 2.5}$	$\mathbf{77.3 \pm 3.5}$
GLM-5.3-Flash	ReAct	77.6 ± 3.3	7.0 ± 3.6	7.4 ± 4.0	45.3 ± 1.5	51.0 ± 4.0
	Reflexion	76.9 ± 2.0	22.7 ± 9.2	23.1 ± 9.6	49.7 ± 3.1	47.7 ± 1.5
	ReCode	93.0 ± 1.9	14.7 ± 22.1	28.5 ± 1.0	55.3 ± 8.3	61.0 ± 8.7
	ReMem	82.6 ± 2.4	67.7 ± 3.5	81.9 ± 2.6	78.3 ± 5.7	66.7 ± 4.6
	SkillAdaptor	86.1 ± 3.5	56.0 ± 3.6	66.6 ± 3.1	85.0 ± 6.0	70.3 ± 2.5
	CPP (Ours)	$\mathbf{99.0 \pm 1.1}$	$\mathbf{69.3 \pm 1.5}$	$\mathbf{83.3 \pm 1.7}$	$\mathbf{95.0 \pm 2.0}$	$\mathbf{77.0 \pm 3.0}$

4 EXPERIMENTS

4.1 SETUP

Environments. We evaluate on ALFWorld, WebShop, MiniGrid, and AppWorld. **ALFWorld** contains text-based household tasks involving navigation and object manipulation; we evaluate on 134 test tasks. **WebShop** evaluates interactive product search and selection, reporting task success rate and reward. **MiniGrid** contains procedurally generated grid-world tasks requiring low-level navigation and object interaction; we evaluate on 100 tasks from 15 categories, using the same tasks across methods. **AppWorld** is included as an additional interactive environment.

Baselines. We compare CPP with ReAct Yao et al. (2022b), Reflexion Shinn et al. (2023), ReCode Yu et al. (2025), ReMem, and SkillAdaptor. ReAct selects actions through interleaved reasoning and interaction; Reflexion uses verbal feedback across attempts; ReCode generates executable code during interaction. ReMem and SkillAdaptor provide additional agent baselines.

Backbone Models. We evaluate GPT-4.1, DeepSeek-V4-Flash, and GLM-5.3-Flash. Backbone model weights remain frozen; each method adapts through its own external interaction mechanism.

4.2 RESULTS

Table 1 summarizes the main results across all environments and backbone models, comparing CPP against the baseline methods in terms of task success and environment-specific evaluation metrics.

CPP delivers consistent gains across all evaluated environments. CPP achieves the strongest overall performance across ALFWorld, WebShop, MiniGrid, and AppWorld under all three backbones. On ALFWorld, for example, CPP reaches $98.5 \pm 1.3\%$, $99.0 \pm 1.1\%$, and $99.0 \pm 1.1\%$ with GPT-4.1, DeepSeek-V4-Flash, and GLM-5.3-Flash, respectively. These results substantially exceed the strongest non-CPP baselines for the corresponding backbones, indicating that the gains are not confined to a particular environment or model family.

The advantage of CPP becomes especially pronounced on challenging long-horizon tasks. The largest improvements appear on MiniGrid and WebShop, where successful behavior requires sustained interaction and effective adaptation over multiple steps. On MiniGrid, CPP obtains $94.3 \pm 2.1\%$,

$96.7 \pm 2.5\%$, and $95.0 \pm 2.0\%$ across the three backbones, compared with the strongest baseline results of $67.7 \pm 7.5\%$, $64.7 \pm 2.5\%$, and $85.0 \pm 6.0\%$, respectively. A similar pattern holds on WebShop: CPP achieves the highest success rate and reward for every backbone, suggesting that its benefit extends beyond task completion to higher-quality interaction trajectories.

CPP transfers robustly across backbones rather than relying on a single strong base model. Despite substantial variation in baseline performance across GPT-4.1, DeepSeek-V4-Flash, and GLM-5.3-Flash, CPP remains consistently strong. This is particularly visible on AppWorld, where CPP scores 76.7 ± 2.5, 77.3 ± 3.5, and 77.0 ± 3.0 across the three backbones, while the strongest baseline performance varies considerably. This suggests that the proposed framework contributes a relatively backbone-independent improvement in agent reliability and generalization.

4.3 Ablations

Table 2 evaluates the contribution of task decomposition and the critic by removing each from CPP.

Both components contribute substantially to overall performance. Removing either component consistently lowers performance across environments and backbones. The effect is particularly large with GPT-4.1, where ALFWorld overall success drops from 98.5% to 59.2% without decomposition and to 30.8% without the critic. Similar degradations on WebShop, MiniGrid, and AppWorld further confirm that both components are important to the effectiveness of CPP.

The two components provide complementary benefits for long-horizon interaction. Task decomposition improves structured execution by breaking complex objectives into manageable subproblems, while the critic supports iterative error correction during execution. Their combination yields the strongest results, indicating that effective planning and feedback are both essential to CPP.

Table 2: Ablation results over three seeds (mean $\pm$ standard deviation). ALFWorld, WebShop SR, MiniGrid SR, and AppWorld SR report task success rate (%); WebShop Reward retains its original benchmark scale. Bold values indicate the highest mean within each backbone and metric.

Backbone	Method	ALFWorld	WebShop		MiniGrid	AppWorld
		Overall SR	SR	Reward	SR	SR
GPT-4.1	**CPP (Ours)**	$\mathbf{98.5 \pm 1.3}$	$\mathbf{65.7 \pm 3.5}$	$\mathbf{79.4 \pm 2.6}$	$\mathbf{94.3 \pm 2.1}$	$\mathbf{76.7 \pm 2.5}$
	w/o Decomposition	59.2 ± 9.4	63.3 ± 4.0	76.5 ± 3.2	69.3 ± 8.0	67.7 ± 4.5
	w/o Critic	30.8 ± 7.5	37.3 ± 3.8	45.4 ± 5.7	56.0 ± 5.6	49.3 ± 2.1
DeepSeek-V4-Flash	**CPP (Ours)**	$\mathbf{99.0 \pm 1.1}$	$\mathbf{66.3 \pm 2.5}$	$\mathbf{78.1 \pm 2.1}$	$\mathbf{96.7 \pm 2.5}$	$\mathbf{77.3 \pm 3.5}$
	w/o Decomposition	86.6 ± 7.9	64.3 ± 1.5	76.2 ± 2.7	75.3 ± 6.1	63.3 ± 4.2
	w/o Critic	88.8 ± 3.4	56.3 ± 3.1	65.2 ± 2.8	59.7 ± 4.5	54.7 ± 4.5
GLM-5.3-Flash	**CPP (Ours)**	$\mathbf{99.0 \pm 1.1}$	$\mathbf{69.3 \pm 1.5}$	$\mathbf{83.3 \pm 1.7}$	$\mathbf{95.0 \pm 2.0}$	$\mathbf{77.0 \pm 3.0}$
	w/o Decomposition	97.5 ± 0.4	63.3 ± 2.9	75.2 ± 2.0	83.3 ± 3.5	63.7 ± 4.2
	w/o Critic	98.8 ± 0.4	61.0 ± 3.6	72.4 ± 2.8	57.7 ± 7.4	55.3 ± 5.5

4.4 Property Evaluations

Patching dynamics. Figure 2 plots the critic's patch-request rate over the task stream. For each episode bin, the rate is the number of critic reviews requesting a patch divided by the total number of critic reviews in that bin. Overall, the critic requests revisions more frequently in the early stages of interaction, with patching rates generally declining over episodes across all four environments. The magnitude and rate of decline vary across environments and models. WebShop maintains relatively high patching rates over much of the interaction, while ALFWorld shows a particularly sharp early decline for DeepSeek V4 Flash. In MiniGrid and WebShop, DeepSeek V4 Flash retains comparatively higher patching rates at later stages, whereas the three models become more similar in AppWorld as interaction progresses. GPT-4.1 also shows a relatively high and persistent patching rate in ALFWorld compared with the other two models.

Cross-task generalization and retention. We evaluate an ALFWorld stream of 10 Pick, 10 Cool/Heat, and 10 further Pick tasks. CPP may revise its policy during the first two phases, but is frozen in the final Pick phase. Table 3 shows high success after the switch from Pick to Cool/Heat: 10/10 tasks for DeepSeek-V4-Flash and 8/10 for GPT-4.1. This supports cross-task generalizability

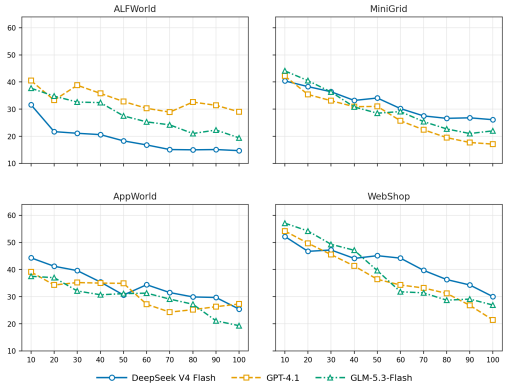

Figure 2: Fraction of critic reviews requesting a patch, by episode bin.

Task-stream phase	DeepSeek	GPT-4.1
Pick (initial)	10/10	9/10
Cool/Heat	10/10	8/10
Pick (policy frozen)	10/10	8/10
Pick SR change (pp)	0	−10

Table 3: Success in the ALFWorld Pick → Cool/Heat → Pick stream (10 tasks per phase; one stream per backbone). DeepSeek denotes DeepSeek-V4-Flash. Policy updates are disabled in the final phase.

in the online adaptation setting. On returning to Pick, the frozen policies succeed on 10/10 and 8/10 tasks, respectively, compared with 10/10 and 9/10 in the initial Pick phase. Thus, the evaluated streams show no observed Pick regression for DeepSeek-V4-Flash and a 10-percentage-point decrease for GPT-4.1. The complete protocol and phase-level patch statistics appear in Appendix B.

Online recovery. Figure 3 shows an ALFWorld episode in which CPP corrects a policy failure without restarting the task. After repeatedly handling a tissue box already at its destination, the critic identifies the faulty object-selection rule. The revised policy instead selects the remaining box and completes the task in four more interactions. Appendix C provides the trajectory and policy change.

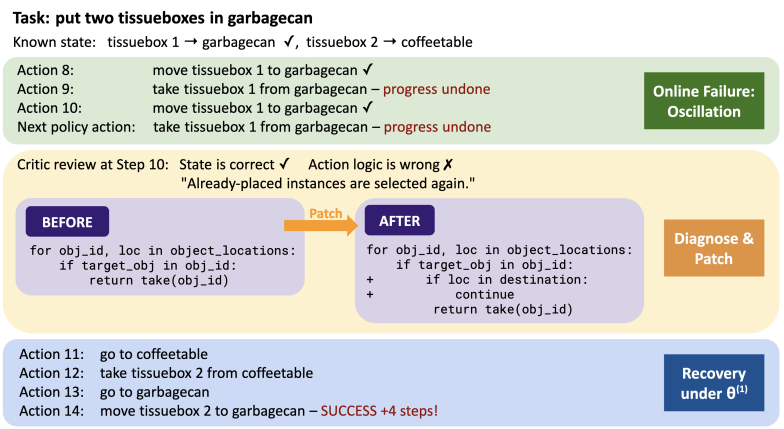

Figure 3: Online recovery within an ALFWorld episode. CPP revises the policy after detecting repeated handling of an already-placed object, then completes the task without resetting the environment.

5 Conclusion

We introduced Continual Policy Programming (CPP), a framework that turns fine-grained online interaction experience into persistent executable policy updates. By interleaving interaction with critique and targeted code revision, CPP allows newly acquired knowledge to affect behavior within the current task while remaining available across future tasks. Across ALFWorld, WebShop, MiniGrid, and AppWorld, CPP enables agents to recover from failures online, generalize learned behaviors across task boundaries, and accumulate capabilities. These results demonstrate that persistent executable policies provide an effective substrate for accumulating behavioral knowledge from streaming experience, enabling continual improvement without updating the underlying LLM.

AI USE STATEMENT

In this work, we used generative AI tools to assist with translation. We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, clean and reformat dataset, support qualitative and thematic data analysis, or interpret results. Additionally, we used generative AI tools to edit the paper to improve its readability. We have reviewed all AI-assisted work. LLM-polished paragraphs were proofread by 2 authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Ashwath Vaithinathan Aravindan, Zhisheng Tang, and Mayank Kejriwal. Code-driven planning in grid worlds with large language models. *arXiv preprint arXiv:2505.10749*, 2025.

Teng Cao, Yu Deng, Hikaru Shindo, Quentin Delfosse, Lanxi Wen, Suli Wang, Jannis Blüml, Christopher Tauchmann, and Kristian Kersting. Kintsugi: Learning policies by repairing executable knowledge bases. *arXiv preprint arXiv:2605.09487*, 2026.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023.

Shiqi Chen, Jingze Gai, Ruochen Zhou, Jinghan Zhang, Tongyao Zhu, Junlong Li, Kangrui Wang, Zihan Wang, Zhengyu Chen, Klara Kaleb, et al. Skillcraft: Can llm agents learn to use tools skillfully? *arXiv preprint arXiv:2603.00718*, 2026.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo Perez-Vicente, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and J K Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *Advances in neural information processing systems*, 36:73383–73394, 2023.

Yupei Du and Dong Nguyen. Measuring the instability of fine-tuning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6209–6230, 2023.

Muhammad Fawi. Curlora: Stable llm continual fine-tuning and catastrophic forgetting mitigation. *arXiv preprint arXiv:2408.14572*, 2024.

Qisheng Hu, Quanyu Long, and Wenya Wang. When continual learning moves to memory: A study of experience reuse in llm agents. *arXiv preprint arXiv:2604.27003*, 2026a.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in llm agents via incremental multi-turn interactions. In *International Conference on Learning Representations*, volume 2026, pp. 156259–156291, 2026b.

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.

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.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Zewen Liu, Zhan Shi, Yisi Sang, Bing He, Minhua Lin, Tianxin Wei, Dakuo Wang, Benoit Dumoulin, Wei Jin, and Hanqing Lu. Adaptive auto-harness: Sustained self-improvement for agentic system deployment on open-ended task streams. *arXiv preprint arXiv:2606.01770*, 2026.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. Clin: A continually learning language agent for rapid task adaptation and generalization. *arXiv preprint arXiv:2310.10134*, 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Haizhou Shi, Zihao Xu, Hengyi Wang, Weiyi Qin, Wenyuan Wang, Yibin Wang, Zifeng Wang, Sayna Ebrahimi, and Hao Wang. Continual learning of large language models: A comprehensive survey. *ACM Computing Surveys*, 58(5):1–42, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *Advances in neural information processing systems*, 36:58202–58245, 2023.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16022–16076. Association for Computational Linguistics, 2024.

Chenxi Wang, Zhuoyun Yu, Xin Xie, Wuguannan Yao, Runnan Fang, Shuofei Qiao, Kexin Cao, Guozhou Zheng, Xiang Qi, Peng Zhang, et al. Skillx: Automatically constructing skill knowledge bases for agents. *arXiv preprint arXiv:2604.04804*, 2026a.

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. *arXiv preprint arXiv:2305.16291*, 2023.

Jiongxiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement learning for self-improving agent with skill library. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1529–1550, 2026b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024b.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H Chi, et al. Evo-memory: Benchmarking llm agent test-time learning with self-evolving memory. *arXiv preprint arXiv:2511.20857*, 2025.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38:17577–17604, 2026.

Chen Yang, Jiashuo Tian, Ziqi Wang, Xinyin Liu, Meiru Ye, and Junjie Chen. Learning globally reusable skills for coding agents. *arXiv preprint arXiv:2608.06153*, 2026.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. WebShop: Towards scalable real-world web interaction with grounded language agents. In *Advances in Neural Information Processing Systems*, volume 35, 2022a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022b.

Zhaoyang Yu, Jiayi Zhang, Huixue Su, Yufan Zhao, Yifan Wu, Mingyi Deng, Jinyu Xiang, Yizhang Lin, Lingxiao Tang, Yuyu Luo, et al. Recode: Unify plan and action for universal granularity control. *arXiv preprint arXiv:2510.23564*, 2025.

Zhuoyun Yu, Xin Xie, Wuguannan Yao, Chenxi Wang, Lei Liang, Xiang Qi, and Shumin Deng. Skilladaptor: Self-adapting skills for llm agents from trajectories. *arXiv preprint arXiv:2606.01311*, 2026.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, 2024.

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.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025a.

Junhao Zheng, Shengjie Qiu, Chengming Shi, and Qianli Ma. Towards lifelong learning of large language models: A survey. *ACM Computing Surveys*, 57(8):1–35, 2025b.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 19724–19731, 2024.

A Experimental Protocol and Reproducibility

Task sets. For ALFWorld, we use all 134 tasks in the `valid_unseen` split. For each of WebShop, MiniGrid, and AppWorld, we sample 100 tasks using the run's random seed. The MiniGrid tasks cover 15 categories. Table 4 summarizes the task sets and interaction limits.

Table 4: Task sets used in the evaluation.

Environment	Task source	Tasks per seed	Max. actions per task
ALFWorld	All `valid_unseen` tasks	134	100
WebShop	Seeded random sample	100	50
MiniGrid	Seeded random sample; 15 categories	100	200
AppWorld	Seeded random sample	100	80

Models and seeds. We evaluate GPT-4.1, DeepSeek-V4-Flash, and GLM-5.3-Flash with frozen model weights. Each experiment uses three different seeds. We first compute the result for each seed, then report the mean and sample standard deviation across the three seed-level results.

CPP implementation. At the start of each task, CPP parses the goal, carries forward the executable policy from the preceding task, and initializes fresh task-local working memory. During interaction, the policy selects actions using the updated working memory and admissible commands. At scheduled review points while the episode is active, the critic examines recent interactions and decides whether to request a revision. A terminal partial window does not trigger an additional review. After a revision, the agent continues from its current environment state. At task termination, the policy persists to the next task, while task-local working memory is discarded.

Each patch replaces three functions together: action selection, working-memory initialization, and working-memory update. The patch may also provide a one-time repair of the live working memory. The implementation specifies these targets as follows:

```
PATCH_TARGETS = (
    ("heuristics", "choose_action", "choose_action_source"),
    ("wm_hooks", "init_working_memory",
     "init_working_memory_source"),
    ("wm_hooks", "update_working_memory",
     "update_working_memory_source"),
)
WM_REPAIR_TARGET = (
    "wm_hooks", "repair_working_memory",
    "repair_working_memory_source",
)
```

The generated functions are loaded into the agent's Python process. The current `admissible_commands` are supplied to action selection, and the returned command is matched against and validated using that list. The validator additionally rejects taking an object while the inventory is nonempty and repeating an action that has already failed at the same location. When a policy function raises an exception during an episode, the runner records the error and selects an admissible fallback action.

Goal parsing receives the task goal and a vocabulary grounded in the current scene. It returns structured target fields and an ordered subgoal plan. For example, the goal-parsing prompt includes the following output requirement:

```
- "subgoal": a SHORT high-level plan for THIS goal --
  ordered steps joined by " -> " describing what
  must actually happen in the environment for the
  task to count as solved.
```

At a review point, the critic receives the task goal, a working-memory summary, recent action–observation transitions, and the current policy source or behavior preview. It assesses progress,

identifies any gap, suggests changes, and ends with a `patch` or `skip` decision. When a patch is requested, the patcher receives this assessment along with the current task context and policy source. Its required response format is:

```
OUTPUT FORMAT (strict):
- Reply with ONE JSON object only.
- Required keys: "choose_action_source",
  "init_working_memory_source",
  "update_working_memory_source"
- Optional key: "repair_working_memory_source"
- Each value: a complete Python function starting
  with the matching `def` line.
```

The optional repair function runs once against the live working memory after the patch is applied; it does not replace the current episode's working memory.

CPP accepts a candidate only after parsing its response, checking the required function definitions and Python syntax, parsing the composed modules, calling the proposed working-memory initializer on a disposable instance, and running functional self-tests. The self-tests check initialization, memory updates, and action selection. A candidate that fails validation is rolled back, and the patcher may make one corrective attempt using the validation error. If that attempt also fails validation, the previous policy remains in effect. One validated and deployed revision counts as one applied patch, even when it replaces multiple functions. Critic decisions to skip and rejected candidates do not count as applied patches.

Baseline configuration. The baseline methods, ReAct, Reflexion, ReCode, ReMem, and SkillAdaptor use the same task sampling procedure as CPP: the entire `valid_unseen` set is used for ALFWorld, and goals are shuffled with the run seed and the first 100 are selected for other environments. Three different seeds are evaluated for each of the backbones.

ReAct (Yao et al., 2022b) generates actions from the task instruction and the current interaction history, using environment-specific few-shot examples. It maintains no persistent state across tasks. Each task is attempted once; the action limit, per-call output-token limit, and handling of invalid actions are specified separately for each environment.

Reflexion (Shinn et al., 2023) augments its action prompt with verbal reflections from earlier attempts at the same task. After an unsuccessful trial, it generates a reflection and starts another trial from the task's initial environment state, up to the configured trial limit. A task that succeeds is not retried. Reflections are retained across trials of that task. Trial limits, action limits, and per-call token limits are specified by environment.

ReCode (Yu et al., 2025) generates executable code that invokes environment-specific action primitives. It constructs a new code tree for each task and may rewrite a failed expansion without restarting the environment. It does not carry the code tree across tasks. The available primitives, expansion limits, environment-action limits, and per-call token limits are specified by environment.

ReMem (Wei et al., 2025) maintains an experience store across tasks. It retrieves relevant prior experiences to inform its reasoning and action selection, and updates the store after a task according to the configured retention rule. Its prompts and action interface are adapted to each environment. The retrieval method, retention rule, internal reasoning limit, environment-action limit, and per-call token limit are specified by environment.

SkillAdaptor (Yu et al., 2026) maintains a collection of reusable skills across tasks. It retrieves relevant skills to guide agent execution, and adapts the skill collection from failed trajectories through step-level failure attribution and targeted skill updates. Candidate skills are retrieved and injected according to task relevance, while proposed revisions are retained only after validation. Its skill initialization, retrieval configuration, adaptation iterations, validation protocol, and agent interaction interface are specified by environment.

Metrics and aggregation. We report task success rate (SR) for ALFWorld, MiniGrid, AppWorld, and WebShop. For a given seed, SR is the percentage of evaluated tasks that satisfy the environment's

success criterion:

$$\mathrm{SR} = 100 \times \frac{\text{number of successful tasks}}{\text{number of evaluated tasks}}.$$

WebShop additionally reports average reward, computed from the environment's per-task reward. For both metrics, an unsuccessful task contributes zero to SR; its contribution to average reward is determined by the WebShop reward returned for that task.

We first compute each metric over the evaluated tasks within each seed, then report the mean and sample standard deviation across the three seed-level values.

B Cross-Task Generalization and Retention: Additional Protocol

This evaluation uses a sequential ALFWorld task stream from the `valid_unseen` split. For each backbone, CPP interacts with 10 Pick tasks, followed by 10 Cool/Heat tasks, followed by 10 Pick tasks. Each task starts with fresh working memory, while the executable policy carries forward to the next task. In the first two phases, the critic reviews the policy regularly and the patcher may deploy a revision. The policy reached after the Cool/Heat phase is frozen for the final Pick phase: neither the critic nor the patcher updates it there. Then the policy is used to evaluate 10 Pick tasks distinct from the initial Pick tasks.

The three phase success rates are reported in Table 3. DeepSeek-V4-Flash succeeds on all 30 tasks. GPT-4.1 succeeds on 9 initial Pick tasks, 8 Cool/Heat tasks, and 8 final Pick tasks. No policy patch is applied in the frozen phase for either backbone.

Switching from Pick to Cool/Heat provides evidence of cross-task generalizability: both backbones maintain high success on the new category while continuing to adapt online. The subsequent frozen Pick phase probes retention of the earlier category after that adaptation. Relative to the initial Pick phase, DeepSeek-V4-Flash shows no observed decrease in success, while GPT-4.1 decreases by one task out of ten. These comparisons concern category-level performance along the observed streams; the initial and final Pick phases are not a paired evaluation of two policy snapshots on the same held-out tasks.

Table 5: Critic requests and applied policy patches in the two adaptation phases of the ALFWorld task stream. Request rates use eligible critic reviews within the corresponding phase as their denominator. The final Pick phase is excluded because policy updating is disabled by design.

Backbone	Phase	Patch requests (% of reviews)	Applied patches
DeepSeek-V4-Flash	Pick	30	9
DeepSeek-V4-Flash	Cool/Heat	10	2
GPT-4.1	Pick	54	21
GPT-4.1	Cool/Heat	59	22

C Online Recovery: Complete Trajectory and Policy Change

We expand the online recovery example in Figure 3. The ALFWorld task requires placing two tissue boxes in a garbage can. The agent finds both instances on `coffeetable 2` and places `tissuebox 1` at the destination. The current policy, however, selects the first instance matching the target class without excluding an instance already placed at the destination. It consequently takes `tissuebox 1` back out and places it again; continuing under the same selection rule would repeat this behavior instead of progressing to `tissuebox 2`.

At the step-10 review, working memory records `tissuebox 1` at the garbage can and `tissuebox 2` at the coffee table. The critic identifies the discrepancy between this stored state and the policy's object-selection branch. The patcher adds a condition that skips target instances whose recorded location is already a valid destination. CPP retains the environment state reached at the review point and continues with the revised policy. Its subsequent actions move toward `tissuebox 2`, take it, return to the garbage can, and place it there. The task succeeds four interactions after the

Table 6: Decisive transitions in the ALFWorld tissue-box episode (`gpt41_syntax_10types_v2`, episode 6). Observations are feedback immediately before the listed action when available. The review occurs after action 10 and does not advance or reset the environment.

Step	Observation or state	Policy action	Outcome
8	`Nothing happens.`	`move tissuebox 1 to garbagecan 1`	First box placed
9	`You move the tissuebox 1 to the garbagecan 1.`	`take tissuebox 1 from garbagecan 1`	First box taken back out
10	`You take the tissuebox 1.`	`move tissuebox 1 to garbagecan 1`	First box placed again
10, review	Inventory empty; `tissuebox 1 @ garbagecan 1; tissuebox 2 @ coffeetable 2.` Policy preview: `take tissuebox 1 from garbagecan 1.`	Critic review and policy patch	Policy revised; no environment reset
11	`You move the tissuebox 1 to the garbagecan 1.`	`go to coffeetable 2`	Moves toward `tissuebox 2`
12	`You arrive at coffeetable 2. On the coffeetable 2, you see a candle 1, a television 1, and a tissuebox 2.`	`take tissuebox 2 from coffeetable 2`	Second box taken
14	`You arrive at garbagecan 1. On the garbagecan 1, you see a tissuebox 1.`	`move tissuebox 2 to garbagecan 1`	Task completed

review. Because this selection rule is part of the persistent policy, it remains available in subsequent tasks. Table 6 records the decisive action–observation transitions from the execution log. The review and patch occur within the same episode; the environment is not reset between the pre-patch and post-patch actions.

The critic's recorded decision and diagnosis are:

> **On track:** no
> **Assessment:** The agent is not making progress toward the task goal of "put two tissuebox in garbagecan." The recent actions show the agent repeatedly moving `tissuebox 1` to `garbagecan 1`, taking it back, and moving it again, without ever attempting to retrieve `tissuebox 2` (which is still at `coffeetable 2`). The working memory correctly records that `tissuebox 1` is in `garbagecan 1`, and `tissuebox 2` is at `coffeetable 2`. The agent's inventory is empty, and it is currently at `garbagecan 1`. The current policy preview is to "take tissuebox 1 from garbagecan 1," which would just repeat the previous oscillation.
> **Decision:** patch

The relevant policy change is the object-selection branch. The same destination check was also inserted into the placement loop. Working-memory update logic was re-emitted with the required three-function rewrite; it was not the change that altered the next action.

```
# Before
# 4. If not holding the object, but know where it is, go take it
if target_obj:
    for obj_id, loc in wm.object_locations.items():
        if target_obj in obj_id and loc != "agent":
            if wm.curr_recep != loc:
                return f"go to {loc}"
            return f"take {obj_id} from {loc}"

# After
# 4. If not holding the object, but know where it is, go take it
```

```
# --- PATCH: Only act on objects NOT already in the destination ---
if target_obj:
    for obj_id, loc in wm.object_locations.items():
        if target_obj in obj_id:
            if dest_recep_ids and loc in dest_recep_ids:
                continue
            if loc != "agent":
                if wm.curr_recep != loc:
                    return f"go to {loc}"
                return f"take {obj_id} from {loc}"
```

Immediately before deployment, the relevant working-memory fields are `wm.inventory = []`, `wm.curr_recep = "garbagecan 1"`, `wm.object_locations["tissuebox 1"] = "garbagecan 1"`, and `wm.object_locations["tissuebox 2"] = "coffeetable 2"`. After deployment they are unchanged: the candidate did not emit a one-time working-memory repair, and these fields already matched the environment. The candidate patch passed validation on the first attempt; the log records a single patch response with `applied: True`. No rollback occurred, and no one-time working-memory repair was performed. The first post-patch action in Table 6 verifies that the revised policy changed behavior in the ongoing episode.

D Review-Interval Sensitivity

Table 7 compares review intervals on ALFWorld across three different W values using DeepSeek-V4-Flash. Both $W = 3$ and $W = 5$ achieve 100% success, but $W = 3$ requires more reviews per task (6.97 versus 4.90) and more applied patches (1.72 versus 0.75), without improving success. Increasing the interval to $W = 20$ reduces reviews to 1.44 and patches to 0.25 per task, but lowers success to 97%. Among these settings, $W = 5$ therefore preserves the highest observed success rate with fewer review and patch operations than $W = 3$.

Table 7: Review-interval sensitivity on ALFWorld using DeepSeek-V4-Flash. Reviews and patches are per task.

W	SR (%)	Reviews	Patches
3	100	6.97	1.72
5 (default)	100	4.90	0.75
20	97	1.44	0.25

E Results by Task Category and Additional Examples

Table 8 contains the category-level success rates of six ALFWorld categories; category sizes differ. The results are evaluated over three different seeds.

Category-level ablations. Table 9 reports the available six-category breakdown for full CPP, w/o decomposition, and w/o critic under each backbone.

Table 8: ALFWorld category success rates (%, mean ± standard deviation over three seeds).

Backbone	Method	Pick	Clean	Heat	Cool	Pick Two	Examine
GPT-4.1	ReAct	81.9 ± 7.1	82.7 ± 4.0	84.1 ± 5.4	79.4 ± 5.9	86.2 ± 2.8	66.7 ± 7.9
	Reflexion	77.8 ± 8.6	87.1 ± 2.6	89.9 ± 4.1	87.3 ± 8.1	82.4 ± 8.3	74.1 ± 5.2
	ReCode	76.4 ± 6.4	79.6 ± 6.7	97.1 ± 2.5	100.0 ± 0.0	100.0 ± 0.0	55.6 ± 14.7
	ReMem	91.7 ± 4.2	79.6 ± 3.7	82.6 ± 0.0	82.5 ± 9.9	76.5 ± 11.8	85.2 ± 8.5
	SkillAdaptor	84.7 ± 7.9	76.3 ± 4.0	79.7 ± 5.4	77.8 ± 2.2	84.3 ± 5.5	87.0 ± 2.6
	CPP (Ours)	98.6 ± 2.0	97.8 ± 3.0	98.6 ± 2.0	100.0 ± 0.0	100.0 ± 0.0	96.3 ± 5.2
DeepSeek-V4-Flash	ReAct	84.7 ± 7.1	90.3 ± 4.6	85.5 ± 2.0	79.4 ± 8.1	80.4 ± 2.8	77.8 ± 4.5
	Reflexion	83.3 ± 3.4	77.4 ± 7.0	91.3 ± 9.4	81.0 ± 7.8	88.2 ± 4.8	83.3 ± 4.5
	ReCode	93.1 ± 4.8	96.8 ± 3.2	97.1 ± 2.5	81.0 ± 12.6	90.2 ± 9.0	81.5 ± 8.5
	ReMem	81.9 ± 6.4	74.2 ± 9.7	78.3 ± 4.3	74.6 ± 12.0	47.1 ± 5.9	92.6 ± 6.4
	SkillAdaptor	90.2 ± 2.0	83.8 ± 2.6	82.6 ± 3.6	81.0 ± 7.8	86.3 ± 5.5	90.7 ± 2.6
	CPP (Ours)	98.6 ± 1.9	98.9 ± 1.5	98.6 ± 2.0	100.0 ± 0.0	100.0 ± 0.0	98.1 ± 2.6
GLM-5.3-Flash	ReAct	76.4 ± 7.9	72.0 ± 5.5	79.7 ± 8.9	79.4 ± 2.2	82.4 ± 4.8	79.6 ± 2.6
	Reflexion	75.0 ± 6.8	73.1 ± 3.0	73.9 ± 6.1	88.9 ± 2.2	80.4 ± 7.3	72.2 ± 0.0
	ReCode	94.4 ± 2.4	94.6 ± 3.7	97.1 ± 5.0	76.2 ± 4.8	100.0 ± 0.0	96.3 ± 6.4
	ReMem	80.6 ± 4.8	76.3 ± 4.9	76.8 ± 9.1	90.5 ± 4.8	78.4 ± 6.8	98.1 ± 3.2
	SkillAdaptor	91.7 ± 3.4	83.9 ± 4.6	79.7 ± 5.4	84.1 ± 5.9	90.1 ± 5.5	88.9 ± 4.5
	CPP (Ours)	98.6 ± 2.0	100.0 ± 0.0	98.5 ± 2.0	98.4 ± 2.2	100.0 ± 0.0	96.3 ± 5.2

Table 9: ALFWorld ablations by task category (%, mean ± standard deviation over three seeds).

Backbone	Variant	Pick	Clean	Heat	Cool	Pick Two	Examine
GPT-4.1	CPP (Ours)	98.6 ± 2.0	97.8 ± 3.0	98.6 ± 2.0	100.0 ± 0.0	100.0 ± 0.0	96.3 ± 5.2
	w/o Decomposition	97.2 ± 2.4	55.9 ± 17.8	58.0 ± 14.0	69.8 ± 15.3	58.8 ± 10.2	3.7 ± 3.2
	w/o Critic	84.7 ± 16.8	16.1 ± 9.7	1.4 ± 2.5	17.5 ± 2.7	70.6 ± 17.6	0.0 ± 0.0
DeepSeek-V4-Flash	CPP (Ours)	98.6 ± 1.9	98.9 ± 1.5	98.6 ± 2.0	100.0 ± 0.0	100.0 ± 0.0	98.1 ± 2.6
	w/o Decomposition	98.6 ± 2.4	93.5 ± 8.5	88.4 ± 13.3	100.0 ± 0.0	90.2 ± 12.2	37.0 ± 21.0
	w/o Critic	97.2 ± 4.8	93.5 ± 3.2	94.2 ± 5.0	96.8 ± 5.5	96.1 ± 6.8	46.3 ± 6.4
GLM-5.3-Flash	CPP (Ours)	98.6 ± 2.0	100.0 ± 0.0	98.5 ± 2.0	98.4 ± 2.2	100.0 ± 0.0	96.3 ± 5.2
	w/o Decomposition	95.8 ± 0.0	97.8 ± 1.9	97.1 ± 2.5	98.4 ± 2.7	100.0 ± 0.0	96.3 ± 3.2
	w/o Critic	100.0 ± 0.0	96.8 ± 0.0	98.6 ± 2.5	100.0 ± 0.0	100.0 ± 0.0	98.1 ± 3.2