CODE-PROBE-ADAPT: INTERACTIVE HEURISTIC LEARNING FOR SELF-EVOLVING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Existing LLM-based autonomous agents generally follow two paradigms: ReAct-style methods, which interleave dense reasoning with continuous API calls; and code-as-policy methods, which synthesize executable code to bypass LLM inference at runtime. However, in long-horizon interactive tasks, ReAct agents incur prohibitive inference costs and context overload, whereas policies revised only after a rollout can delay adaptation to unexpected behavior. To bridge this gap, we introduce **Code-Probe-Adapt** (COPA), an interactive heuristic learning framework that combines efficient code execution with timely inspection and revision. Its persistent REPL (Read-Eval-Print Loop) harness enables live, on-demand introspection and short tests from the reached state. To enable adaptive code-as-policy, COPA modulates policy execution using adaptive step budgets and event-triggered interruptions. The triggering condition helps focus inspection, while live probing provides evidence for policy refinement. Experiments on five games demonstrate the effectiveness and interaction efficiency of COPA. On Montezuma's Revenge, its event-triggered pauses and live probing support targeted route and action-timing tests before episode completion, achieving an average score of 4,900 in 2.4M steps while Heuristic Learning and free-form Codex score 0 at 6M steps. Removing live probing or automatic interruption reduces the score to 160 and 400, respectively. On Pong and Breakout, COPA matches both agent baselines' maximum scores using over $11\times$ fewer environment steps. Attained entirely without gradient updates or task-specific engineering, COPA successfully unites the execution efficiency of code with the adaptive power of live debugging. Project page: https://copa-code-probe-adapt.pages.dev/.

1 INTRODUCTION

The rapid advancements in Large Language Models (LLMs) have catalyzed the emergence of autonomous agents capable of perceiving, reasoning, and acting within complex environments (Xi et al., 2025; Wang et al., 2024b). Existing LLM-based autonomous agents generally follow two interaction paradigms, contrasted with COPA in Fig. 1. The first is ReAct-style online control, which interleaves dense LLM reasoning with continuous environment actions at every step (Yao et al., 2023). This design endows the agent with continuous adaptability, a capability that subsequent frameworks have augmented with memory, reflection, and experiential learning (Park et al., 2023; Shinn et al., 2023; Madaan et al., 2023; Zhao et al., 2024; Wang et al., 2024a). However, for tasks requiring complex reasoning, particularly in long-horizon interactive environments, the necessity of querying the LLM at every step incurs prohibitive inference costs, high reaction latency, and rapid context window saturation. Furthermore, the inherent stochasticity of LLMs means that highly frequent interventions often introduce compounding erroneous perturbations, ultimately leading to suboptimal task performance in dynamic settings (Paglieri et al., 2025; Lin et al., 2025).

To circumvent the bottlenecks of dense reasoning, a second paradigm has emerged: code-as-policy. In this approach, LLMs generate executable programs that bypass model inference entirely during the decision-making phase (Liang et al., 2023; Wang et al., 2024a;c; Gao et al., 2023; Chen et al., 2023). This shifts the LLM's role from a direct, step-by-step decision-maker to a high-level programmer that interacts with the environment via synthesized code. Recent systems further demonstrate the promising potential of code-driven agent self-evolution, as evidenced by automated AI research frameworks (Karpathy, 2026), AutoHarness (Lou et al., 2026), alongside works on heuristic learning

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.

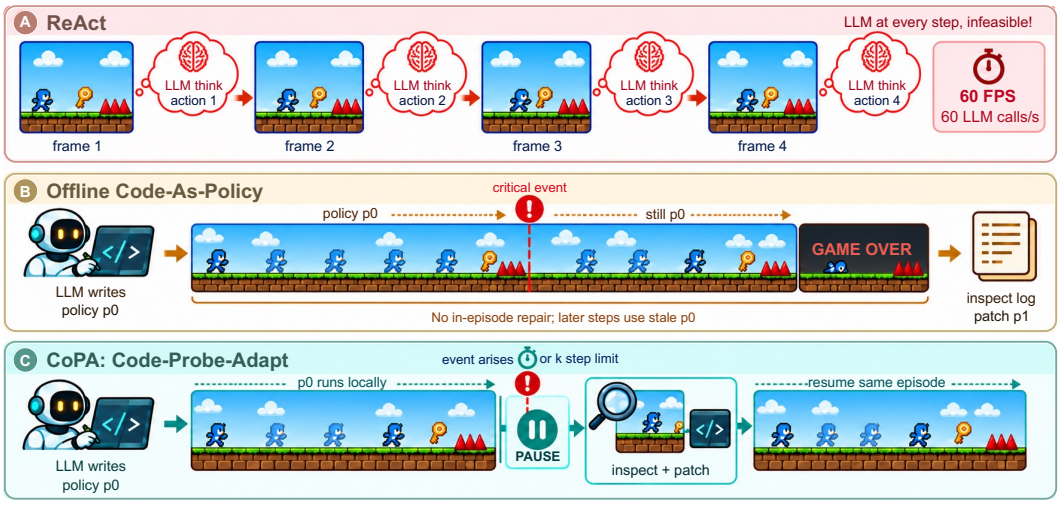

Figure 1: **Motivation for Code-Probe-Adapt: efficient execution with timely, focused adaptation.** (A) ReAct-style agents adapt through per-step LLM reasoning, incurring high inference costs, latency, and context overload. (B) Offline code-as-policy executes efficiently but defers policy revision until the rollout ends, which can waste steps after a heuristic fails. (C) CoPA combines autonomous execution with sparse interventions. Event triggers focus live analysis on critical states, enabling timely, targeted probing and policy refinement within the same episode.

(Weng, 2026). By compiling an agent's behavior into code, these methods capitalize on the speed and determinism of executable programs, proving highly efficient and precise for low-level control routines. However, in offline code-as-policy, a program's control logic remains fixed throughout a rollout, limiting adaptation to unforeseen situations not covered by the current heuristics. Coding agents can revise these heuristics through post-hoc log analysis, but doing so only after a rollout ends can waste environment steps. The resulting logs may still leave unclear when a failure began or which assumption needs revision. A zero return, for example, may reflect a poor route, incorrect target recognition, or mistimed actions; further reflection on the same record may not distinguish them. This makes the timing and focus of intervention important for learning efficiency: **when should the agent pause execution, what should it inspect, and which short continuation would help determine what to revise?** Thus, timely, focused access to diagnostic evidence can improve policy learning while retaining the execution efficiency of code.

To bridge this gap, we introduce **Code-Probe-Adapt** (CoPA), an interactive heuristic learning framework in which the agent designs policies that execute action sequences autonomously between sparse interventions. To this end, CoPA introduces agent-defined segment boundaries governed jointly by a dynamic step limit k and self-configured event triggers (e.g., life loss or score changes). The agent adjusts k based on environmental feedback, shortening it when closer inspection is needed and extending it as policy behavior stabilizes. Execution pauses when either condition is met: event triggers help identify what to inspect, while a persistent REPL (Read–Eval–Print Loop) preserves the live environment for targeted probing and policy revision. For example, in *Montezuma's Revenge*, the agent can inspect the position reached along a proposed route to a key and test a short continuation to assess whether the route or action timing needs revision. This enables policy revision before episode completion and avoids the per-action LLM calls of ReAct-style control.

To support these interventions and prevent information loss during LLM-absent phases, CoPA uses auto-tracking memory to preserve task-relevant variables. For example, the agent can inspect a ball's position and velocity instead of a dense sequence of video frames. Notably, CoPA is **task-agnostic by design**: it uses the same agent instructions across all games, with task-specific policies developed through environmental interaction.

Our main contributions are as follows. (1) We propose CoPA, an interactive heuristic learning framework equipped with a persistent REPL harness, enabling LLM agents to inspect live execution evidence, test short continuations, and revise policies within an ongoing episode. (2) We introduce

agent-defined segment boundaries for adaptive code-as-policy: a dynamically adjusted step limit and self-configured event triggers jointly determine when autonomous execution pauses for agent intervention. Auto-tracking memory preserves task-relevant information, while the triggering condition helps focus inspection on behavior needing diagnosis. (3) Empirically, we show the value of timely, focused access to diagnostic evidence for policy discovery. On Montezuma's Revenge, CoPA enables the agent to inspect critical states and test short continuations, turning ambiguous zero-reward failures into evidence for route and action-timing repairs. It achieves an average score of 4,900 in 2.4M steps, while Heuristic Learning and free-form Codex score 0 at 6M steps; removing live probing or automatic interruption reduces the score to 160 and 400, respectively. On Pong and Breakout, CoPA reaches the same maximum scores as both agent baselines using over $11\times$ fewer environment steps (Table 1).

2 Methodology

CoPA enables an LLM agent to develop executable policies through online interaction with a black-box environment. The policies act autonomously between sparse interventions, while agent-defined segment boundaries determine when execution pauses for inspection and revision. A persistent REPL preserves the live environment at each pause, allowing the agent to probe uncertain behavior and refine its policy before episode completion (Fig. 2).

2.1 Problem Formulation and System Overview

We formulate each task as a black-box interaction problem with an observation space $\mathcal{O}$ and a discrete action space $\mathcal{A}$. During autonomous execution, the executable policy receives an observation $o_t \in \mathcal{O}$ and selects an action $a_t \in \mathcal{A}$ at each step t. The public interaction interface also provides rewards, episode termination signals, and environment metadata e_t. We enforce the black-box setting by blocking the agent's access to simulator internals, offline demonstrations, and hand-crafted task-specific strategies. Given a limited interaction budget, the objective is to improve task performance by synthesizing and revising executable policies; task-specific metrics are defined in Sec. 3.

System Architecture. CoPA consists of an LLM policy designer and a persistent REPL harness. The policy designer serves as a unified coding and reasoning agent: it formulates hypotheses, writes policy code, and interprets execution evidence. It also maintains durable memory files that record observations and lessons from previous interactions. The harness maintains the live environment and executes the current policy locally, without per-step LLM calls. Execution proceeds in segments rather than requiring the completion of an episode: at each segment boundary, control returns to the agent while the reached environment state remains available for inspection. The agent can then revise the policy and resume interaction from that state. Adaptation occurs through changes to code and memory, without updating model weights.

2.2 Adaptive Code-as-Policy with Sparse Intervention

To support both autonomous execution and timely intervention, CoPA organizes the generated policy as $\Pi = \langle M, \pi, \Phi \rangle$:

- **Auto-Tracking Memory (M):** Agent-defined tracking logic, specified through `track_memory`, preserves task-relevant information across steps, such as object positions and inferred velocities. We denote the tracked memory at step t by M_t. This memory supports policy execution and subsequent inspection without requiring the LLM to process every raw observation.
- **Executable Policy (π):** A Python function maps the current observation, tracked memory, and public metadata to an action, $a_t = \pi(o_t, M_t; e_t)$. The agent encodes and revises its task-specific heuristics in this function.
- **Critical Event Trigger (Φ):** An agent-defined `stop_event_fn` monitors the observable state and tracked memory for conditions that warrant inspection, such as life loss or a score change. When a condition is met, it signals the harness to interrupt execution and may provide a reason for the interruption.

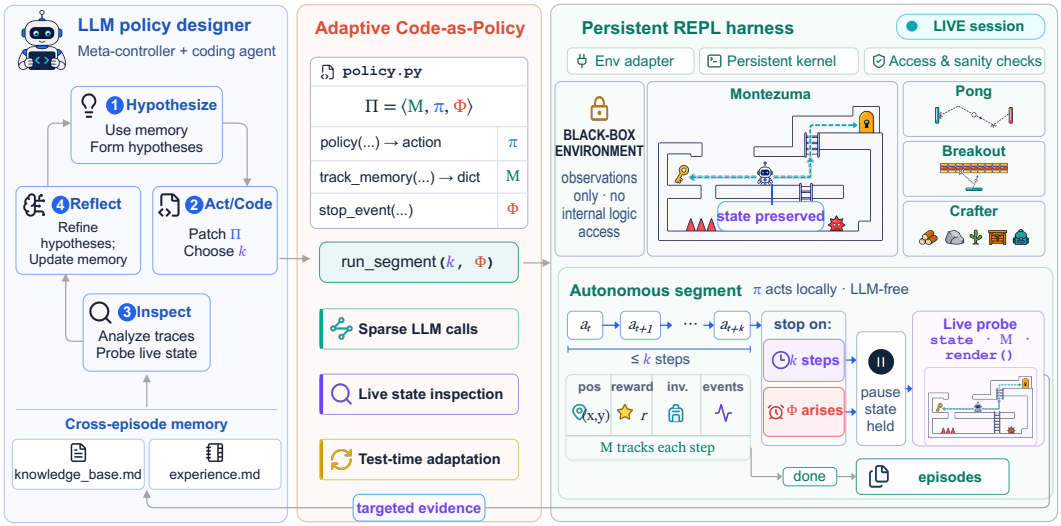

Figure 2: **Overview of Code-Probe-Adapt (CoPA).** CoPA combines the efficiency of code-as-policy with test-time adaptability by invoking the LLM only between autonomous segments. The policy designer patches $\Pi = \langle M, \pi, \Phi \rangle$ and selects an execution horizon k. The persistent REPL then executes π locally without LLM calls for up to k steps; during execution, M records task-relevant variables, while Φ interrupts the segment upon detecting a critical event. Crucially, the live session preserves the reached state at each pause, enabling targeted probes instead of post-hoc analysis of dense trajectories. The resulting evidence updates the policy and cross-episode memory, forming a gradient-free loop of autonomous execution, live inspection, and test-time adaptation.

Agent-Defined Segment Boundaries. For each invocation of `run_segment`, the agent specifies a step budget k. The harness executes the policy π, updates memory, and evaluates event triggers Φ entirely locally—avoiding per-step LLM inference—until k steps elapse, Φ detects a critical condition, or the episode terminates. Control then returns to the agent to inspect outcomes, revise π, and calibrate k for the next segment. By shortening k when closer oversight is needed and lengthening it as behavior stabilizes, the agent dynamically modulates intervention frequency, while Φ interrupts execution when the specified conditions are detected. Beyond avoiding unnecessary execution, the triggering condition guides diagnosis by identifying the event and associated variables to inspect. Event triggers preserve recognized critical states, while a finite horizon also allows inspection of unexpected behavior outside the trigger rules, such as repeated left–right motion without reward or life loss. Adaptive k supports short exploratory probes and longer execution of established behavior.

2.3 Interactive Probing through a Persistent REPL

The persistent REPL makes each pause an opportunity to investigate the behavior that led to the reached state. The environment and session variables remain available between calls, so the agent can inspect the current observation, tracked memory, recent rewards, and traces before deciding how to proceed. This allows diagnostic probing and policy revision within an ongoing episode, without reconstructing the situation from a terminal rollout log.

During inspection, the agent uses the triggering condition, when present, and tracked memory to select evidence relevant to its current hypothesis. For example, it can examine a tracked ball position and velocity, compare recent rewards, or call `render()` to inspect the current frame. If the cause remains ambiguous, it can execute a short continuation from the preserved state to gather additional evidence before deciding which policy assumption to revise. To avoid context overload, sub-agents process raw observations and dense traces and return concise semantic summaries to the agent.

During an intervention, the optional clone/restore interface lets the agent test alternative action sequences from a saved state and then continue or restore.

2.4 Operational Loop: Hypothesize, Act/Code, Inspect, Reflect

CoPA repeats four stages to turn interaction feedback into executable policy updates: (1) **Hypothesize**: use the current observations, tracked memory, and durable notes to state what the policy should try next or why its previous behavior failed; (2) **Act/Code**: encode the hypothesis through `set_policy_source`, select k and any event trigger, and execute the policy through `run_segment`; (3) **Inspect**: examine targeted evidence from the paused session and perform additional probes when needed; (4) **Reflect**: compare the observed outcome with the expected behavior, record the conclusion through `reflect`, and decide which parts of the policy or hypothesis to retain or revise. If the current policy remains informative, the agent can continue observing it with a revised step budget or trigger, without changing its action logic.

To retain lessons across interventions, the agent maintains two durable memory files:

Knowledge Base (`knowledge_base.md`). The knowledge base distinguishes active hypotheses from observations supported by interaction evidence. When a probe supports a hypothesis, the agent records the relevant evidence, such as rewards, step counts, or observed state changes. It revises or removes earlier conclusions when subsequent observations contradict them.

Experience Bank (`experience.md`). The experience bank records lessons from successful and failed policy executions, including behaviors to reuse and mistakes to avoid. For example, a route that reaches a key can inform subsequent policy revisions, while a failed jump can motivate a shorter segment or an additional event trigger.

These files preserve accumulated knowledge for later interventions, while M_t provides the tracked information used during autonomous execution. Cross-episode retention follows the evaluation protocol: Atari discovery carries lessons across episodes within a run, whereas each Crafter episode begins with freshly initialized policy and memory states (Sec. 3.2). When an episode ends, the agent records its outcome and starts a new episode if the protocol allows further interaction. The process continues until the task's stopping criterion is met or the interaction budget is exhausted.

3 Experiments

Our experiments test the central claim of CoPA: sparse LLM interventions, when paired with live state inspection and code-level policy repair, can yield executable policies that adapt at test time instead of relying on dense per-step reasoning. Atari provides visually structured control tasks for evaluating whether CoPA can discover and repair executable policies through interaction. Montezuma's Revenge further stresses this mechanism under sparse rewards, where the agent must probe alternative continuations at uncertain states and convert the successful ones into reusable policy code. Crafter then evaluates whether the same CoPA loop supports within-episode adaptation in an unfamiliar open-world environment, where the agent revises its policy from its own observations, rewards, and reflections. Tables 1, 3, and 2 summarize the resulting score-versus-interaction evidence, Crafter adaptation score, and averaged Atari discovery accounting.

Experimental configuration. CoPA, Heuristic Learning (Weng, 2026), and free-form Codex use GPT-5.5 with the reasoning effort set to `xhigh` and the same environment settings. Codex chooses its own execution, inspection, and revision schedule; CoPA adds its persistent REPL harness and adaptive segment boundaries. In the main Atari comparison, the agent controls the environment solely through action execution. All Atari discovery runs are capped by the runner at 6M steps, except the extended Montezuma experiment, which uses 12M. We compare agent-based methods; published RL results provide context.

Task-agnostic protocol. Across all games, CoPA uses the same agent instruction template for policy discovery, probing, and adaptation (Appendix C). Task-specific policies are developed through environmental interaction, without game-specific strategy instructions.

Table 1: **Atari scores and environment steps.** CoPA entries are discovery-run means. The 200M-step RL reference scores are digitized from the Atari learning curves in Hafner et al. (2023), matching Fig. 3; the second row for each RL baseline gives its 400K-step result. – denotes unreported values.

Method	Pong		Breakout		Montezuma	
	Score	Steps	Score	Steps	Score	Steps
Human	15	–	30	–	4753	–
Random	-21	0	2	0	0	0
DreamerV3 (Hafner et al., 2023)	20.4	200M	383.0	200M	1853.5	200M
400K setting	-4.3	400K	9.3	400K	–	–
PPO (Schulman et al., 2017)	18.5	200M	294.5	200M	2283.8	200M
400K setting	-20.3	400K	2.6	400K	–	–
Heuristic Learning	21	1.49M	864	4.99M	0	6.0M
Codex	21	758K	864	4.60M	0	6.0M
CoPA (ours)	**21**	**61.8K**	**864**	**411.2K**	**4900**	**2.4M**
CoPA + clone/restore	–	–	–	–	5900	4.0M

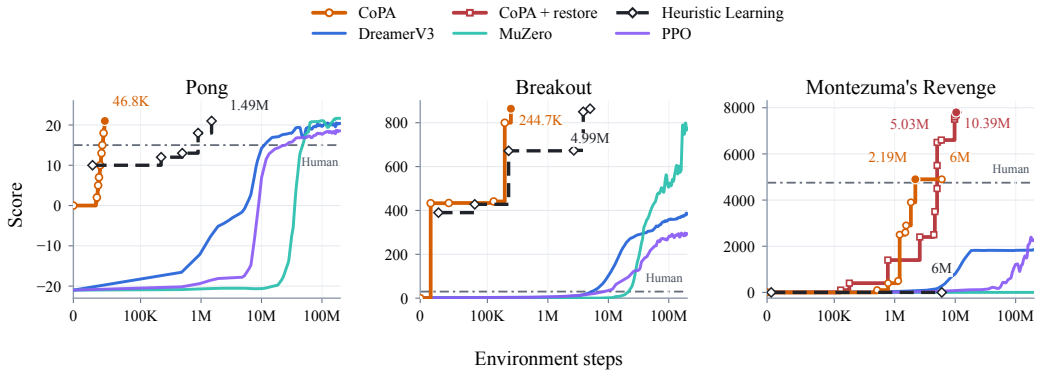

Figure 3: **Representative Atari discovery traces with RL references.** Discovery traces show best-so-far scores; the restore-enabled Montezuma trace is the extended run. Published RL training curves are included as references.

3.1 Atari: Probe-to-Best Policy Discovery

Protocol. For Atari, discovery serves a role analogous to training: the agent searches for high-reward policies and routes through interaction, without gradient updates. At each intervention, the agent forms a hypothesis about the game state, edits `policy.py`, adjusts the step budget k and event trigger Φ, observes the next rollout segment, and reflects on the resulting reward, failure mode, and trace evidence. For each Atari game, we conduct five independent CoPA discovery runs. We average each run's best episode score from its retained combination of segment-specific policies and the costs incurred up to first attaining that score. Our Heuristic Learning reproduction remains at zero on Montezuma, so its reported costs cover the 6M-step setting. Environment steps include all executed branches, including discarded trials.

Frozen-policy evaluation. We evaluate the retained policy combination from each CoPA discovery run under three random seeds, with policy code and persistent memory files fixed and clone/restore disabled. Each combination reproduces its best discovery score under all three seeds.

Policy-discovery trajectories. Figure 3 shows rapid gains on Pong and more gradual progress on Breakout. On Montezuma, standard CoPA plateaus while the restore-enabled agent improves further.

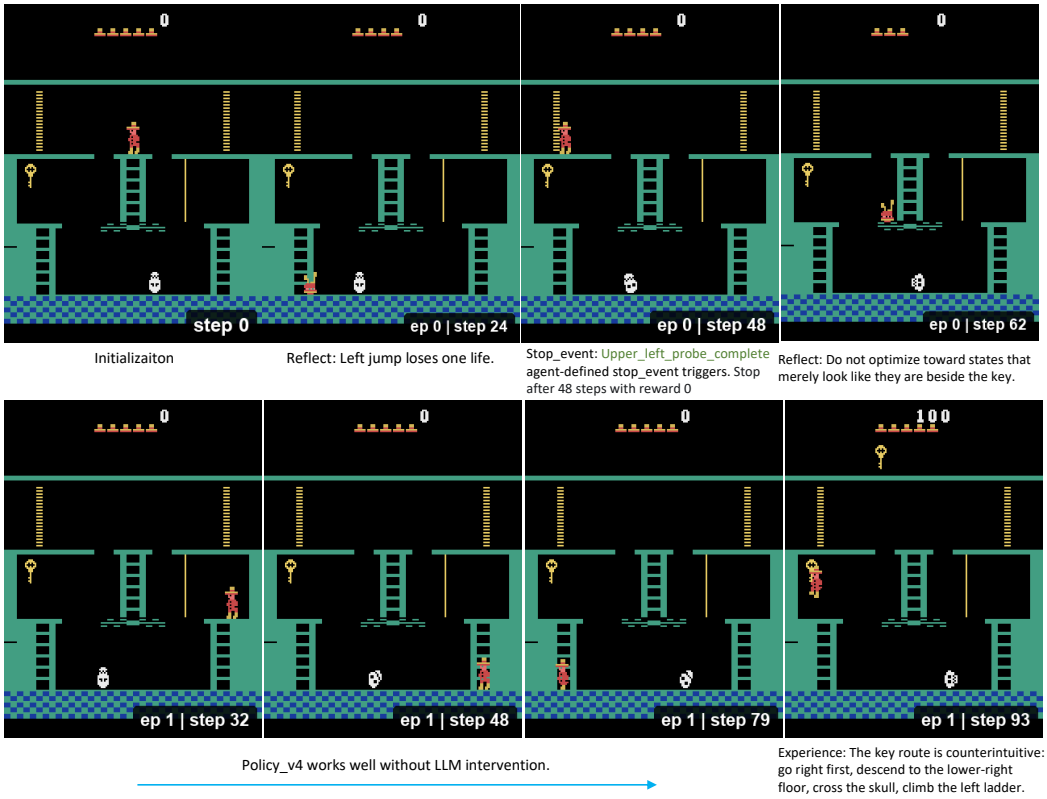

Figure 4: **How CoPA discovers the first key in Montezuma's Revenge.** The agent tests route hypotheses through autonomous policy execution and reflects on feedback at segment boundaries. After the upper-left probe yields no reward, it revises the policy to follow the lower route to the key and records the successful route in the experience bank for reuse.

Case study: Montezuma's first key. Figure 4 illustrates how CoPA tests route hypotheses through policy execution and uses segment-level feedback to guide policy revision. The initial policy reached an upper-left ledge that appeared close to the key, but an agent-defined event stopped the probe with zero reward. Reflecting on this outcome, the agent reconsidered the route and revised the policy to test a lower path. The revised policy executed the action sequence to the key autonomously, and the agent recorded the successful route in the experience bank for reuse. This process allows the agent to evaluate and revise its strategy during exploration, without waiting for episode completion or requiring per-action LLM calls.

Pong. CoPA averages the maximum Pong score of 21 at 61.8K environment steps in Table 1. The representative curve in Fig. 3 shows that this performance emerges in fewer than 100K environment interactions, making Pong a test of rapid visual-controller synthesis. The final policy is a compact visual controller: it detects the white ball and right paddle from RGB frames, predicts the ball trajectory, tracks the paddle center, and adds a directional contact bias to create offensive return angles. This result shows that the agent can rapidly turn trace-level evidence into a stable executable control heuristic when the perceptual structure is accessible.

Breakout. CoPA averages a Breakout score of 864 at 411.2K environment steps in Table 1. The representative curve in Fig. 3 shows a different regime from Pong: score gains follow repeated repairs as the screen distribution shifts from dense racks to sparse endgame states. The score reaches the two-rack maximum of 864 in 411.2K steps, compared with 4.99M for Heuristic Learning and 4.60M for Codex (Table 1), requiring approximately $12.1\times$ and $11.2\times$ fewer steps, respectively. This result illustrates how CoPA accumulates task-specific control logic through successive policy updates.

Table 2: **Atari agent discovery costs.** LLM tokens count uncached input plus output. Costs end at first attainment of the reported score, or at 6M steps for zero-score runs. – denotes unreported values.

Task	Method	Score	Env. steps	LLM tokens	Policy versions	LLM interventions
Pong	Heuristic Learning	21	1.49M	168K	7	–
	Codex	21	758K	216K	–	–
	CoPA (ours)	**21**	**61.8K**	**304.1K**	**9**	**19**
Breakout	Heuristic Learning	864	4.99M	789K	56	–
	Codex	864	4.60M	1M	–	–
	CoPA (ours)	**864**	**411.2K**	**449.0K**	**129**	**21**
Montezuma	Heuristic Learning	0	6.00M	6.99M	118	–
	Codex	0	6.00M	3.8M	–	–
	CoPA (ours)	**4900**	**2.40M**	**4.67M**	**286**	**344**
	CoPA + clone/restore	5900	4.00M	6.39M	237	301

Montezuma's Revenge. Montezuma's Revenge tests exploration under sparse rewards. CoPA averages 4900 points after 2.4M environment steps (Table 1), while Heuristic Learning and Codex score 0 at 6M steps. Event-triggered pauses preserve critical states for live inspection and short continuation tests, helping the agent identify which route or action-timing assumption to revise.

Standard CoPA is more interaction-efficient early in discovery but plateaus around 4900 points, while clone/restore supports further improvements. Independent clone/restore runs average 5900 points at 4.0M steps. To test whether additional interactions allow this variant to improve further, we increase its discovery budget to 12M steps. The extended run reaches 7800 points after 10.39M steps (excluded from the reported means). We observe the agent returning to saved states to explore alternative continuations, resembling Go-Explore's return-then-explore strategy (Ecoffet et al., 2021).

Cost Accounting. Each pause that returns control to the LLM counts as one intervention, encompassing the inspection and any action tests performed during that pause. Table 2 compares agent discovery scores, environment steps, and LLM tokens (Appendix A.6).[1] On Pong, CoPA reaches the same maximum score using approximately 24× fewer environment steps than Heuristic Learning and 12.3× fewer than Codex. Appendix A.5 analyzes the token–interaction trade-off.

Component Ablations. Without live probing or automatic interruption, Montezuma scores drop from 4,900 to 160 and 400, respectively (Appendix A.2). Adaptive k alone yields 2,500 points and event triggers alone yield 2,900; fixed $k = 1000$ with event triggers reaches 4,200. The full method reaches 4,900 with 20% fewer steps than the fixed-horizon setting (2.4M versus 3.0M), but uses more LLM submissions (344 versus 280; Appendix A.3).

Additional visual-control task. On VizDoom d3, a 3D first-person task, CoPA scores 606.4 versus 546.6 for Heuristic Learning, with best-run scores of 924 and 668, respectively (Appendix A.4).

3.2 Crafter: Test-Time Adaptation without Pre-Probing

Protocol. Crafter evaluates whether the same policy-repair loop transfers beyond Atari-style control. While the original Crafter benchmark evaluates learning over 1M environment steps (Hafner, 2022), our protocol targets zero-shot within-episode adaptation. We evaluate 10 episodes with different random seeds, each with freshly initialized policy and memory states and no separate Crafter training or pre-probing phase. The environment provides RGB observations and public adapter metadata such as inventory, newly unlocked achievements, and a local semantic view; game-specific strategies, recipes, walkthroughs, and handcrafted subgoals are excluded. The agent follows the same sparse intervention loop: propose an executable hypothesis, let the policy act for a segment, inspect rewards and failures, and revise `policy.py`. Thus, the protocol evaluates zero-shot exploration and online

[1] The original Heuristic Learning work reports a Montezuma mean of 133.3 across three discovery runs and a best score of 400 (Weng, 2026).

policy repair within individual episodes. We use the official achievement-score formula to summarize adaptation performance in Table 3:

$$\text{Score} = \exp(\bar{g}) - 1, \qquad \bar{g} = \frac{1}{N}\sum_{j=1}^{N} \log(1 + 100 p_j),$$

where $N = 22$ and $p_j \in [0, 1]$ is the fraction of the 10 evaluation episodes in which achievement j is unlocked at least once.

Results. CoPA achieves an aggregate Crafter score of 19.2, and Codex scores 17.6, each over 10 independent episodes. The RL scores in Table 3 provide context under different interaction budgets and observation interfaces. Across the 10 episodes, the observed behaviors span a broad early-to-mid-game skill set, including tool crafting, table and furnace placement, resource gathering, eating, combat, and iron sword crafting. These behaviors emerge through sparse policy edits between rollout segments, illustrating adaptation through iterative code repair.

Table 3: **Crafter score comparison.** Scores use the benchmark achievement-success formula in Sec. 3.2. Both agent results aggregate 10 independent episodes.

Agent / Reference	Crafter score
Human expert	50.5 ± 6.8
Random	1.6 ± 0.0
PPO ResNet (Moon et al., 2023)	15.6 ± 1.6
DreamerV3 (Hafner et al., 2023)	14.5 ± 1.6
DreamerV2 (Hafner et al., 2021)	10.0 ± 1.2
PPO (Schulman et al., 2017)	4.6 ± 0.3
Codex	17.6
CoPA (ours)	19.2

Interpretation. The key signal is within-episode adaptation under uncertainty: the agent infers useful mechanics from ongoing interaction and repairs its policy after failures. Trace notes identify recurring repair targets: food and drink recovery, hostile encounters during ore routes, and post-furnace planning.

4 Discussion and Conclusion

CoPA reframes LLM agency as interactive heuristic learning rather than dense action prediction. The LLM maintains executable hypotheses, code handles routine environment steps, traces provide evidence, and probes test local uncertainty. This division of labor is the central result: useful test-time learning can happen through source-level policy repair without updating model weights.

The experiments support this claim in two complementary regimes. In visually structured Atari tasks, sparse interventions are sufficient to build high-performing policies when the policy can track the right state variables. In Montezuma's Revenge, the agent inspects failures, falsifies hypotheses, and preserves successful repairs across later execution; in Crafter, the same inspect-and-repair loop operates within each episode. These results position CoPA between program synthesis, classical planning, and deep reinforcement learning, with learned artifacts represented as editable programs.

The long-term opportunity is a dual-system agent: learned perception and control where data is abundant, plus symbolic debugging and code adaptation where novelty matters.

Limitations

CoPA relies on the underlying LLM to synthesize and debug executable policies, interpret sparse feedback, and maintain hypotheses across revisions. Less capable models may fail to recover from brittle heuristics or erroneous policy revisions.

Training or distilling a local model for trace interpretation, reflection, and policy repair could reduce this dependency while preserving the code-probe-adapt loop.

AI USE STATEMENT

Generative AI tools were used to assist with language editing and LaTeX formatting of this manuscript. All AI-assisted changes were reviewed by the authors, who take responsibility for the final content, including its text, claims, citations, and artifacts.

REPRODUCIBILITY STATEMENT

Section 2 specifies the method and its interfaces, while Sec. 3 documents the experimental settings, interaction accounting, and evaluation results. Appendix C provides the complete master prompt used by the agent.

REFERENCES

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *TMLR*, 2023.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return, then explore. *Nature*, 2021.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *ICML*, 2023.

Danijar Hafner. Benchmarking the spectrum of agent capabilities. In *ICLR*, 2022.

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering Atari with discrete world models. In *ICLR*, 2021.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv*, 2023.

Andrej Karpathy. autoresearch: AI agents running research on single-GPU nanochat training automatically. GitHub repository, 2026.

Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, 2016.

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 *ICRA*, 2023.

Wenye Lin, Jonathan Roberts, Yunhan Yang, Samuel Albanie, Zongqing Lu, and Kai Han. GAMEBoT: Transparent assessment of LLM reasoning in games. In *ACL*, 2025.

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*, 2026.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *ICLR*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

Seungyong Moon, Junyoung Yeom, Bumsoo Park, and Hyun Oh Song. Discovering hierarchical achievements in reinforcement learning via contrastive learning. In *NeurIPS*, 2023.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wołczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktäschel. BALROG: Benchmarking agentic LLM and VLM reasoning on games. In *ICLR*, 2025.

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. In *UIST*, 2023.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv*, 2017.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. ProgPrompt: Generating situated robot task plans using large language models. In *ICRA*, 2023.

Dídac Surís, Sachit Menon, and Carl Vondrick. ViperGPT: Visual inference via Python execution for reasoning. In *ICCV*, 2023.

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. *TMLR*, 2024a.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024b.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *ICML*, 2024c.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *ICLR*, 2025.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. In *NeurIPS*, 2023.

Jiayi Weng. Learning beyond gradients. Blog post, 2026.

Yue Wu, Shrimai Prabhumoye, So Yeon Min, Yonatan Bisk, Ruslan Salakhutdinov, Amos Azaria, Tom Mitchell, and Yuanzhi Li. SPRING: Studying the paper and reasoning to play games. In *NeurIPS*, 2023.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 2025.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *AAAI*, 2024.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the Minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv*, 2023.

A Additional Experimental Results

A.1 Crafter Interaction Reward

During interaction, the raw reward is

$$R_i = \sum_{t=1}^{T_i} r_{i,t}, \qquad r_{i,t} = |\mathcal{U}_{i,t} \setminus \mathcal{U}_{i,t-1}| + 0.1\,(h_{i,t} - h_{i,t-1})\,,$$

where $\mathcal{U}_{i,t}$ is the set of achievements unlocked by time t and $h_{i,t} \in \{0, \ldots, 9\}$ is health.

A.2 Component Ablations

Table 4 compares the full framework with four component ablations. The *w/o Auto-Interrupt* variant removes both the per-segment step limit k and the event trigger Φ. Every variant reaches the maximum Pong and Breakout scores, though each ablation uses more environment interactions. On Montezuma, the largest score drops occur without live probing or automatic interruption, falling from 4,900 to 160 and 400, respectively.

The live-probing ablation uses more interactions (4.6M versus 2.4M steps) yet reaches only 160 points, supporting the value of live inspection. The following subsection further ablates k and Φ separately.

Table 4: **Component ablations on Atari.** Full CoPA denotes the standard configuration.

Variant	Pong		Breakout		Montezuma	
	Score	Steps	Score	Steps	Score	Steps
Full CoPA	**21**	**61.8K**	**864**	**411.2K**	**4900**	**2.4M**
w/o REPL (live probing)	21	613.6K	864	3.5M	160	4.6M
w/o Auto-Interrupt	21	524.4K	864	1.5M	400	3.6M
w/o External Memory	21	186.2K	864	800.9K	2900	3.6M
w/o Auto-Tracking Memory	21	80.8K	864	706.3K	5000	3.2M

A.3 Adaptive Horizons and Event Triggers

Ablation setup. Table 5 compares four settings: adaptive k only (Φ disabled), event triggers only (no finite segment horizon), fixed $k = 1000$ with Φ, and the full method. Episode termination and the overall interaction budget apply throughout.

Table 5: **Horizon and event-trigger ablations on Montezuma.** LLM interventions count REPL submissions, which may batch multiple local segments. – denotes unreported token costs.

Variant	Score	Env. steps	LLM interventions	LLM tokens
Adaptive k only	2500	4.2M	246	–
Event trigger only	2900	4.9M	224	–
Fixed $k = 1000$ + event trigger	4200	3.0M	280	–
Adaptive k + event trigger	**4900**	**2.40M**	**344**	4.67M

Adaptive intervention. The full method achieves the highest score with the fewest environment steps among these settings. Relative to fixed $k = 1000$ with Φ, it improves the score by 16.7% and reduces environment steps by 20.0%, with more LLM submissions (344 versus 280). The two controls support complementary aspects of live diagnosis: Φ stops execution at recognized critical events, while adaptive k creates opportunities to inspect unexpected behavior and test short continuations. Their combination supports timely policy repair as exploration progresses.

Fixed horizons. We use $k = 1000$ as a fixed-horizon reference. Fixed-only exploration with short horizons was limited by LLM cost. Small horizons can require frequent LLM inspection, while large horizons can pass informative failure states before inspection, making initial reward discovery harder. CoPA lets the agent choose k within a general framework, adapting from short exploratory probes to longer execution as policy behavior stabilizes.

Execution diagnostics. Table 6 illustrates this adaptation in one full-method run: long route executions are interleaved with short local probes, and life loss is the most frequent stopping event, directing inspection toward failed attempts.

Table 6: **Execution diagnostics from one Montezuma discovery run.** Left: selected horizons and executed lengths in chronological order. Right: stopping reasons across 10,355 local segments through the first 4,900-point result, including preliminary exploration.

Probe	Cell	Requested k	Steps
Initial probe	2	45	40
Route to 2,600	204	1574	1569
Local continuation	300	194	189
Local inspection	325	85	80
First 4,900	329	2514	2485

Stopping reason	Count	Share
Life loss	7314	70.63%
Script completion	2693	26.01%
Episode termination	102	0.99%
Reward	92	0.89%
Room change	90	0.87%
Position condition	27	0.26%
No event note	37	0.36%

A.4 VizDoom d3

We also evaluate on VizDoom d3 (Kempka et al., 2016), a 3D first-person visual control task. CoPA achieves a score of 606.4 compared to 546.6 for Heuristic Learning, with best-run scores of 924 and 668, respectively (Table 7).

Table 7: **VizDoom d3 results.** Best-run scores are reported separately from task scores.

Method	Score	Best-run score
Heuristic Learning	546.6	668
CoPA (ours)	**606.4**	**924**

A.5 Intervention–Cost Trade-Off

Table 8 compares two Breakout discovery runs that both reach 864. The run with more LLM interventions consumes more LLM tokens but requires fewer environment steps. These observations illustrate a trade-off between token usage and environment interactions; they do not isolate the causal effect of intervention frequency. Neither resource count measures wall-clock or monetary cost.

Table 8: **Comparison of two Breakout discovery runs, both reaching a score of 864.**

Run	Score	LLM interventions	LLM tokens	Env. steps
1	864	15	314K	577K
2	864	26	583K	244K

A.6 Agent Baselines and Cost Accounting

Heuristic Learning, free-form Codex, and CoPA run in Codex with GPT-5.5 and reasoning effort set to `xhigh`. All three use the same Atari settings: single RGB frames of 210×160 pixels, frame skips of 1, 1, and 4 for Pong, Breakout, and Montezuma, respectively, no reward clipping or sticky actions,

and public RAM access only on Montezuma. Each main discovery run has a 6M-step budget; the extended Montezuma run uses 12M. Heuristic Learning follows its policy-synthesis and evaluation workflow; Codex chooses its own controller, execution, inspection, and revision schedule. CoPA supplies the persistent REPL harness, adaptive segment boundaries, and memory interfaces described in Sec. 2. In Crafter, both agents use 10 independent episodes with freshly initialized policy and memory states. They receive RGB observations and the same public inventory, achievement, and local semantic fields, use the native 17-action space and reward, and have no separate training or pre-probing phase (Sec. 3.2).

Environment steps include all executed trials and probes. Discovery costs end when the reported score is first attained; zero-reward runs are reported at the 6M-step budget. LLM tokens are uncached input plus output; cached input is excluded, and reasoning tokens are already part of output. Policy versions count newly submitted control programs, without expanding candidates generated inside a submitted search program.

B Related Work

Table 9: **Qualitative comparison of agentic adaptation mechanisms.** Columns denote sparse LLM calls, executable code used as the policy, adaptation from feedback at test time, a persistent REPL-style execution context, automatic improvement of the maintained artifact, automatic interruption of autonomous execution, and automatic tracking of task-relevant state.

Method	Sparse Call	Code-as Policy	Test-Time Adaptation	REPL	Auto Evolve	Auto Interrupt	Auto Track
ReAct (Yao et al., 2023)	✗	✗	✓	✗	✗	✗	✗
Reflexion (Shinn et al., 2023)	✗	✗	✓	✗	✓	✗	✗
Voyager (Wang et al., 2024a)	✓	✓	✓	✗	✓	✗	✗
CodeAct (Wang et al., 2024c)	✗	✗	✓	✓	✗	✗	✗
AutoHarness (Lou et al., 2026)	✓	✓	✗	✗	✓	✗	✗
Heuristic Learning (Weng, 2026)	✓	✓	✗	✗	✓	✗	✗
CoPA (ours)	✓	✓	✓	✓	✓	✓	✓

LLM agents for games and long-horizon interaction. ReAct established a paradigm wherein language models interleave reasoning traces with external actions (Yao et al., 2023). Subsequent game and embodied agents extend this framework through self-reflection, hierarchical planning, environmental feedback, and executable skill libraries (Shinn et al., 2023; Madaan et al., 2023; Zhao et al., 2024; Wang et al., 2024a; Zhu et al., 2023; Wu et al., 2023; Wang et al., 2023). Voyager shares similar motivations, employing an automatic curriculum, a growing library of executable code skills, and iterative prompting with environment feedback for open-ended learning (Wang et al., 2024a). These systems demonstrate that LLMs can refine behavior by integrating observations, failures, and prior experience. CoPA focuses on controlling the boundary between autonomous policy execution and LLM intervention: the agent defines when a persistent policy pauses, then uses evidence from the live session to inspect and revise its behavior. Table 9 summarizes these distinctions.

Code actions, code policies, and synthesized harnesses. Executable code functions both as a computational medium for reasoning and as an action space for agents. Frameworks like PAL and Program-of-Thought delegate computation to Python interpreters (Gao et al., 2023; Chen et al., 2023), whereas CodeAct adopts executable Python as a unified action space for multi-turn LLM agents (Wang et al., 2024c). In embodied domains, Code as Policies and ProgPrompt synthesize programmatic robot plans from language prompts (Liang et al., 2023; Singh et al., 2023), with related systems leveraging code for visual reasoning and reward design (Surís et al., 2023; Ma et al., 2024). Furthermore, AutoHarness demonstrates that LLMs can synthesize game-specific code harnesses—and occasionally entire policies—to mitigate illegal actions and reduce inference-time queries (Lou et al., 2026). CoPA couples a persistent policy module with agent-defined pauses and live-state probing, allowing policy repair within an ongoing episode.

AutoResearch and Heuristic Learning. Recent work highlights a paradigm shift from updating neural weights to maintaining executable heuristic systems. Karpathy (2026) introduces an autonomous research loop in which an agent iteratively refines a training script, executing fixed-budget experiments and retaining modifications that improve validation performance. Similarly, Weng (2026) conceptualizes "Heuristic Learning" as a framework where a coding agent interprets multimodal feedback (e.g., rewards, execution logs, and video traces) to directly edit policies, state detectors, and memory modules. CoPA builds on this maintenance view by linking policy revision to agent-defined segment boundaries and inspection of the preserved environment state.

Interactive software agents and REPL interfaces. Modern coding agents increasingly operate within persistent development environments, leveraging interactive shells, unit tests, execution traces, and patch cycles to resolve complex software engineering tasks (Yang et al., 2024; Wang et al., 2025; Qiao et al., 2023). This progression underscores the importance of interfaces that expose rich, high-level abstractions rather than constraining the model to low-level, step-by-step execution. CoPA adapts this debugging paradigm to agent control: the environment harness acts as an interactive test runner, where rollouts, memory summaries, rendered observations, and diagnostic probes provide empirical evidence to systematically refine the agent's working policy hypothesis.

C Prompt Template

The master prompt from `PROMPT.md` is reproduced below.

Master prompt from PROMPT.md

```
# Test-Time Heuristic Learning via Adaptive Code-as-Policy

You are an autonomous agent discovering an optimal policy at test time through
black-box interaction with `{ENV_ID}`. A persistent Python kernel owns the live
environment. You interact with it in a REPL style: submit Python cells, inspect
the live state, then decide the next action.

## 0. Run Context
- `WORK_DIR=/path/to/GameAgentOutput`
- `ENV_ID=crafter`
- `RUN_DIR={WORK_DIR}/{ENV_ID}_{DATE}_{TIME}` All run artifacts, including
temporary cell files, must stay under `RUN_DIR`.

---

## 1. Constraints
- **Black Box**: Rely only on observations, rewards, done, info, `env_meta`,
frames, traces, `RUN_CARD.md`, and `env_describe.md`.
- **Forbidden**: Do not read simulator source, game-specific solution code, ROMs,
or perform game-specific web searches/walkthroughs.
- **Independence**: Do not ask the human for help. Never stop until the
theoretical maximum reward is achieved.
- **Efficiency**: Treat environment steps/episodes as precious resources. Do not
use loops to batch-run segments blindly. Tune $k$ to the suitable steps needed for
feedback (smaller $k$ during exploration; larger $k$ after the policy behavior is
stable, and use `stop_event_fn` as a guard against wasting steps after critical
events).

---

## 2. Public Cell API
Read api files (`session.py`, `kernel.py`, `cell_protocol.py`, `run.sh`,
`envs/{ENV_ID}.py`).
Interact with the kernel using the following Python API:
- `set_policy_source(src: str, hypothesis: str) -> dict`: Compiles, saves, and
registers the active policy source.
```

```
- `run_segment(k: int, stop_event_fn=None) -> dict`: Runs up to `k` steps.
`stop_event_fn` is used for early stop when critical event happens.
- `reflect(expected: str, observed: str, hypothesis: str, reflection: str,
next_action: str, verdict: str = "N/A")`: Appends a segment-level reflection log.
- `start_episode()`: Resets/starts the next episode once the current one ends.
- `status() -> dict`: Returns game status.
- `render(path: str)`: Saves the current frame as an image.
- `finalize_run(verdict: str, final_report_md: str, commit_msg: str, shutdown:
bool)`: Saves summary logs and terminates the kernel.
- `session`: raw state, memory, trace, recorder frames, and last `info` (Do NOT
access private variable).

---

## 3. Discovered Policy Specification
`policy.py` must define:
1. `policy(state, memory, env_meta) -> action`
2. `track_memory(env_meta) -> dict` (Optional: defines persistent state tracking.
For example: position/state/relationship of the objects,)
3. `stop_event_fn(state, memory, env_meta) -> Union[None, bool, str, dict]`
(Optional: stops segment early when matching specific conditions, returning a
reason/dict)

---

## 4. Execution Loop (Iterate per Segment)

For each cycle, execute these steps sequentially:

### Step 1: Hypothesize
- Formulate a concrete hypothesis about what the policy should try next or why the
previous behavior failed.

### Step 2: Act/Code
- Implement the hypothesis via `set_policy_source(src, hypothesis)`.
- Choose an appropriate $k$ and call `run_segment(k)`.

### Step 3: Inspect
- Inspect the return value of `run_segment`, `session.state`, `memory`,
`session.recorder.frames[-result["steps"]:]` or latest `info`.
- Use `render()` to visually inspect states if necessary.
- Delegate long state/log analysis to sub-agents or temporary helper functions.
Run them locally to digest raw data, and only output the final semantic
conclusions to preserve your main context window.
### Step 4: Reflect & Update Memory
After every segment, perform the two actions:


#### 4.1: Maintain Persistent Memory Files
Read, update, and refine these two files in `RUN_DIR` to inform your next
decisions:

1. `knowledge_base.md`:
   - **Hypotheses**: List active, unproven assumptions.
   - **Validated Knowledge**: Once a hypothesis is proven, move it here and append
   concrete **Fact Support** (evidence, step counts, or values). Append other
   useful knowledge here.
   - **Updates**: Actively correct/edit existing knowledge if new evidence proves
   it wrong. Avoid duplicate entries.

2. `experience.md`:
   - Log specific successful or failed execution experiences (concrete lessons
   learned, dos, and don'ts).
```

````
#### 4.2: Call `reflect(...)` with the following arguments:
- `observed`: Critical observations. What failed or improved.
- `hypothesis`: [Validated | Partially | Refuted | N/A].
- `reflection`: Deep-dive analysis on why the outcome occurred. What should be
tracked next? Any new hypothesis emerged? Reusable lessons?
- `next_action`: Next adjustment/test step.

After reflection, return to Step 1 with the next hypothesis, or keep observing if
the current policy remains informative. Adjust the $k$ value, stop_event_fn, and
memory tracking if needed.

### Step 5: Handle Episode End (If `episode_ended == True`)
1. Append an episode summary to `notes.md` using this format:
   ```markdown
   ## ep N (reward=R, cum_steps=S)
   [Summary of the episode: strategies tried, key observations, and next episode
   priorities]
   ```
2. Call `start_episode()` to initialize the next run.

### Step 6: End Loop (ONLY if theoretical highest reward is achieved)
Call `finalize_run(...)`.

---

## 5. Bootstrap
1. If `RUN_DIR` does not exist: Initialize the directory/git repo, create `tmp/`,
start the kernel, and wait for `.kernel_ready`.
2. Read `env_describe.md` and `RUN_CARD.md`.
3. Begin the Execution Loop.
````