NEURO-SYMBOLIC COMPUTER USE: LEARNING REUSABLE POLICIES FOR RELIABLE AND EFFICIENT EXECUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Many computer tasks recur: the same workflow runs many times, with new inputs and from different starting states. Current computer-use agents re-plan every step of every run, which makes them costly and unreliable on such tasks. We introduce *neuro-symbolic computer use*, in which a recurring workflow is executed by a learned policy rather than re-derived by an agent on each run. The policy fixes the decisions that are stable across runs (ordering, variables, loops, and branches) in executable code, and delegates observation-dependent decisions, such as grounding and state checks, to neural models. We learn these policies with *neuro-symbolic policy iteration*: starting from one agent trajectory, it executes the policy, diagnoses failures with task-completion and step-level judges, and revises the code with a coding model informed by an agent's continuation from the point of failure, without access to the benchmark evaluator. Iterating on generated parameter and initial-state variants makes the policy reusable, and a pre-action verifier guards each state-mutating step at deployment. On OSWorld-Verified and ScienceBoard, the learned policies achieve the highest Passˆ3 of all methods in all four settings, 3.6–15.8 points above the base agent, while cutting per-run cost by 15–217$\times$ and latency by 3.4–5.1$\times$. On OSWorld-Verified, policies built only on variants transfer to the held-out original tasks, exceeding AutoRPA by 8.6–17.5 points in Passˆ3.

> *Civilization advances by extending the number of important operations which we can perform without thinking about them.*
>
> — Alfred North Whitehead

1 INTRODUCTION

A person who files expense reports a thousand times does not rediscover the same procedure on the thousandth attempt. However, contemporary computer-use agents still do: they plan step-by-step on every run, as if it were the first (Agashe et al., 2025a;b; Gonzalez-Pumariega et al., 2026b; Sun et al., 2025; Xie et al., 2024). For economically-valuable recurring workflows (Patwardhan et al., 2026) where the same algorithmic procedure only needs limited decision-making, agents that plan from scratch on every step of every run face two critical problems: unreliability over repeated runs of the same task (Gonzalez-Pumariega et al., 2026a), and high cost and latency of frontier-model calls on long workflows (Yuan et al., 2026). How can an agent turn execution experience into a reusable procedure that stays reliable and efficient as inputs and states vary?

Existing ways to automate such workflows sit at two extremes. At the neural extreme, computer-use agents decide every action from the current observation (Agashe et al., 2025b; Gonzalez-Pumariega et al., 2026b; Qin et al., 2025; Wang et al., 2025a). They are flexible, but re-decide steps that never change, so every run is costly and lacks reliability. At the symbolic extreme, robotic process automation (RPA) runs a recorded or hand-written script (Van der Aalst et al., 2018), which is cheap to repeat but breaks when a states changes and does not generalize beyond anticipated cases.

To solve this dilemma, we introduce the paradigm of **Neuro-Symbolic Computer Use**, in which a recurring workflow is executed by a learned neuro-symbolic policy instead of being re-derived by an

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.

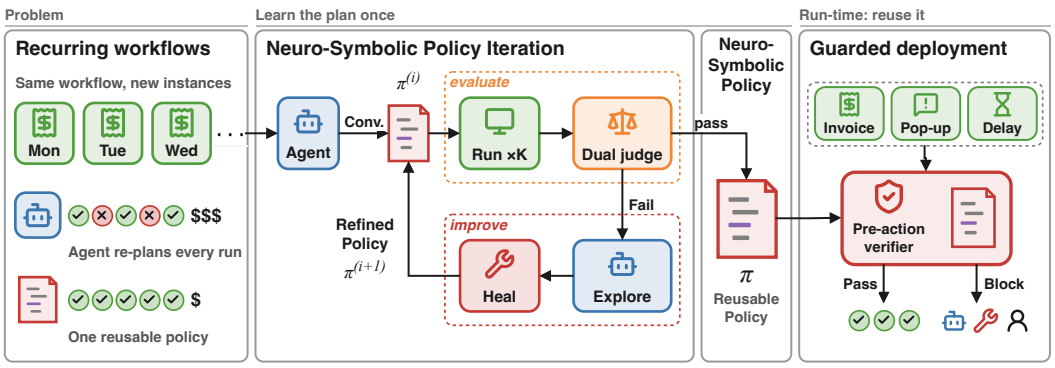


Figure 1: *Recurring* computer workflows require reliable repeated execution with generalizability at low cost. In *Neuro-Symbolic Policy Iteration*: a baseline agent's exploration seeds an initial neuro-symbolic policy, which is iteratively judged, diagnosed and improved with additional agent continuation, then guarded with pre-action verifiers during test-time deployment.

agent on every run. Our core insight is that recurring workflows contain a mixture of deterministic transitions and state-dependent decisions, and that neuro-symbolic programs are reliable and efficient *representations* of such workflows. In expense reporting for example, form fields almost never change, but the location of a "submit" button may vary and a check for login may be needed. We therefore represent the procedure as a neuro-symbolic policy: code fixes the stable decisions (algorithm, variables, and control flow), while neural primitives make state-dependent decisions at run time.

To learn such policies, we introduce **Neuro-Symbolic Policy Iteration (NSPI)**, which synthesizes an initial policy from a base neural agent's demonstration trajectories and refines it by iterating between policy evaluation and policy improvement without ground truth rewards. A task-completion judge checks the end state of each iteration's policy rollout, and a step-level judge locates the first step that went wrong. Upon a failure verdict, an agent continues the task from the failed state, and a coding model revises the policy using this continuation and the judges' diagnosis. This execute–diagnose–explore–revise loop repeats until every rollout in an iteration passes or the budget runs out. At convergence, the policy itself represents a summary of the learned experience, and only here does it receive feedback from the benchmark evaluator. To test generalizability we also refine it on task variants, before its deployment along with a pre-action verifier guarding each state-mutating action against environment drift.

We evaluate NSPI on three properties that recurring workflows require: reliability across repeated runs, reuse on similar task instances, and efficiency. On OSWorld-Verified (henceforth OSWorld) (Xie et al., 2024) and ScienceBoard (Sun et al., 2025) with GPT-5.6 Terra and Claude Opus 5 as base agent models, the learned policies achieve the highest Pass^3 (the share of tasks solved in all three runs) of all methods in all four settings, 4.5–13.9 points above the base agent. We also learn policies only on synthetic OSWorld variants that change a task's parameters or its starting state, and evaluate them on the held-out original tasks, where they exceed AutoRPA (Chen et al., 2026) on the same task families by 8.6-17.5 points in Pass^3. Because a learned policy does not re-derive the workflow, it runs 15–217× cheaper and 3.4–5.1× faster than the base agent, and the one-time cost of learning it is recovered after 3–27 runs.

2 Neuro-Symbolic Computer Use

2.1 Problem Statement: Recurring Workflows and Reusable Policies

We consider recurring parametric computer tasks on open-world desktop environments, a common setting in personal logistics and business processes. A task g is a natural-language instruction to be completed in a digital environment (a desktop application or a website) with transition function T and initial state distribution ρ. Executing a policy π from an initial state $s_0 \sim \rho$ produces a trajectory $\tau = (o_0, a_0, o_1, \ldots, o_{|\tau|})$ of observations and actions. A held-out success indicator $R(\tau; g) \in \{0, 1\}$ (e.g., the evaluation script of a benchmark or a user's unknown preferences) determines whether



Figure 2: Example: conversion of agent trajectory and healing of a neuro-symbolic policy with state-dependent control flow, task parameters in code, and neural primitives for grounding (Section 4.1).

τ completed g. We require a policy for each g to succeed across a parameterized family $\mathcal{G}$ of task instances similar to g (e.g., "book a flight on {date}") and across initial states $s_0 \sim \rho$.

Objective and reliability metric. We seek a policy that maximizes reliability over the task family:

$$\max_{\pi} \ \mathbb{E}_{g\sim\mathcal{G}}\big[\text{Pass\^{}k}(\pi; g)\big], \tag{1}$$

where Pass^k (Yao et al., 2024) is the probability that all k independent executions of π on g succeed:

$$\begin{aligned} \text{Pass\^{}k}(\pi; g) &:= \mathbb{P}\big[R(\tau_1; g) = \cdots = R(\tau_k; g) = 1\big] = p_\pi(g)^k, \\ p_\pi(g) &:= \mathbb{E}_{s_0\sim\rho,\, \tau\sim(\pi,T)}\big[R(\tau; g)\big], \end{aligned} \tag{2}$$

Given n executions with c successes, $\binom{c}{k}/\binom{n}{k}$ is the unbiased estimator of Pass^k for any $k \le n$.

Amortized cost metric. Let C_π be the one-time cost (in dollars or wall-clock time) of constructing π and c_π its cost per execution; the amortized cost per execution over n executions is

$$\bar{c}_\pi(n) := \frac{C_\pi}{n} + c_\pi. \tag{3}$$

A pure neural agent μ has $C_\mu = 0$ (ignoring the costs of model training) but a high c_μ if it calls an expensive LLM to plan every step. In general, a policy π with $C_\pi > 0$ and a lower c_π would have lower amortized cost once n exceeds the break-even point $n^* = C_\pi/(c_\mu - c_\pi)$.

2.2 Neuro-symbolic policies

A neuro-symbolic policy is a program P in a scripting language whose statements are control flow (loops, conditionals, variable assignment, and recursion) and calls to primitives from a set $\mathcal{A} = \mathcal{A}_{\text{sym}} \cup \mathcal{A}_{\text{neu}}$ (detailed in Section A.1). We write Π for the class of policies defined by such programs. Symbolic primitives in $\mathcal{A}_{\text{sym}}$, such as keyboard shortcuts, shell commands, and file operations, execute without a model call. Neural primitives in $\mathcal{A}_{\text{neu}}$ call a model on the current observation o_t, and our work includes: 1) GUI grounding actions (e.g., click(description)) map natural-language element descriptions to screen coordinates using a visual grounding model; 2) a condition check $Q(\phi, o_t) \in \{0, 1\}$, e.g., stateSatisfies(condition) $\to \{0, 1\}$, returns whether o_t satisfies a natural-language condition ϕ, and the control flow of P can branch on its output. All neural primitives include the accessibility tree and the screenshot in o_t as input by default, so the policy only needs to represent natural language inputs. Figure 2 shows an example policy before and after refinement. At deployment, the program runtime applies each state-mutating primitive with the pre-action verifier of Section 3.2.2.

3 Neuro-Symbolic Policy Iteration

NSPI first learns a policy that solves a task reliably (Section 3.1) and then generalizes it beyond the instance it was learned on (Section 3.2); Figure 1 shows both parts.

3.1 POLICY LEARNING

3.1.1 SYNTHESIS AND ITERATIVE REFINEMENT

Policy iteration. We search Π for a policy that maximizes Equation (1) by policy iteration (Howard, 1960; Sutton & Barto, 2018), which alternates between policy evaluation and policy improvement. A dual judge evaluates the policy (Section 3.1.2), and a code healer improves it (Section 3.1.3). Together, the two judges are the analog of the critic in policy iteration. Despite not being fit to ground-truth rewards like a classical critic, they rely on the knowledge of the underlying LLMs to produce an evaluation of the current policy in natural language.

Initialization. Given a task g, a base agent μ produces a trajectory τ_μ, which need not be successful. A converter Φ_{conv} turns τ_μ into the initial policy $\pi^{(0)}$: it concatenates the agent's actions into a single program, replaces pixel coordinates with natural-language descriptions of the target elements that the runtime grounds at execution time, and replaces values specific to g with task parameters.

Iteration. At iteration i, we execute $\pi^{(i)}$ up to K times, each from a freshly reset initial state, and judge each rollout $\tau_j^{(i)}$ with the dual judge of Section 3.1.2 (policy evaluation). If all K rollouts pass the judge, we return $\pi^{(i)}$. Otherwise, we stop at the first failed rollout and produce $\pi^{(i+1)}$ with the code healer of Section 3.1.3 (policy improvement). The loop runs for at most N iterations and returns the last policy if no iteration passes. Note again that R is not used; we apply it only the returned policy to measure Pass^k. Appendix A.3 gives further details of the loop.

3.1.2 POLICY EVALUATION WITH A DUAL JUDGE

We evaluate each policy rollout with two judges to detect and localize failures and provide diagnosis.

Task-completion judge. The task-completion judge J_{task} is a VLM agent that receives only the task g (and the answer submitted by the policy, for question-answering tasks) and probes the end state of the machine with read-only tools, such as shell commands and file reads. It returns a verdict $v_{\text{task}} \in \{\text{success}, \text{failure}, \text{abstain}\}$ with the evidence e_{task} it collected, and it abstains when success depends on transient states during task execution; an applicability gate skips it for tasks whose success cannot be checked from persistent state.

Step-level judge. The step-level judge runs on every rollout, in parallel with J_{task}, and has two stages. First, for each transition t, an observer VLM describes the change between consecutive observations, $\delta_t = J_{\text{obs}}(o_t, o_{t+1})$, without access to the task, the code, or the stated intent of the step, so that each description is based only on observable evidence. Second, a localizer VLM reads the task, the policy, and all observer outputs, and returns a verdict, the first step at which the rollout diverged from the task, and a named failure mode, $(v_{\text{loc}}, t^*, m) = J_{\text{loc}}(g, \pi, \delta_{0:|\tau|-1})$.

Verdict and diagnostic. A rollout passes only if v_{loc} is success and v_{task} is success or J_{task} was skipped by the applicability gate, so an abstention by J_{task} counts as a failure. An aggregator LLM composes the outputs of both judges, $(v_{\text{task}}, e_{\text{task}})$ and (v_{loc}, t^*, m), into a diagnostic d for the healer. We measure how often the judges agree with the benchmark evaluation scripts in Section 4.7. Algorithm 1 states the procedure, and Appendix A.4 gives the details of both judges.

3.1.3 POLICY IMPROVEMENT WITH A CODE HEALER

When rollout $\tau_j^{(i)}$ fails, the base agent μ continues the task from the end state of $\tau_j^{(i)}$, given g, the policy $\pi^{(i)}$, its execution log, and the diagnostic d, and produces a continuation τ_{cont}. A summarizer LLM Φ_{sum} condenses it into a short description of the actions that made progress. The healer Φ_{heal}, a coding LLM, then rewrites the policy:

$$\pi^{(i+1)} = \Phi_{\text{heal}}\big(\pi^{(i)}, \tau_j^{(i)}, d, \Phi_{\text{sum}}(\tau_{\text{cont}}); g\big), \tag{4}$$

where $\tau_j^{(i)}$ enters as its execution log, error message, and a subset of its screenshots. Upon execution errors, we skip the judges and agent continuation, giving the error directly to the healer.

3.2 POLICY GENERALIZATION

NSPI generalizes to anticipated task variations by running policy iteration on generated variants that perturb the task's parameters and initial state, so that the healer must write code that reads the parameters of the instance and handles each initial state (Section 3.2.1). Section 4.3 tests generalization to held-out tasks. Unanticipated environment nonstationarity are caught at deployment by a pre-action verifier, which can hand the task to a general agent or escalate it for human review in sensitive domains such as healthcare and finance (Section 3.2.2).

3.2.1 ROBUST LEARNING UNDER WORKFLOW PERTURBATIONS

Given a target task g, we construct variants of g in two categories of workflow variation, run policy iteration on the variants, and hold out g itself for evaluation (Figure 7). **Initial-state variants** keep the instruction of g and start the environment in a different realistic state, e.g., the target document is closed, a dialog blocks the window, or the output file already holds stale values. We run the base agent on natural language instructions to set up such initial states. **Parameter variants** keep the initial state of g and change the argument values in its instruction, e.g., a date, a count, a file name, or a setting value. An LLM proposes multiple variants per category, and a validator rejects any that changes the goal of the task or the content of taks setup files. Task variants are presented to NSPI sequentially, thereby exposing hard-coded values and unhandled states, forcing the policy to avoid overfitting any single variant. Section 4.1 describes the variants we construct for each benchmark.

3.2.2 PRE-ACTION VERIFICATION

We check the precondition (Fikes & Nilsson, 1971) of every state-mutating primitive using a fast and calibrated VLM classifier before deploying the final policy (Figure 6 in Appendix A.5).

Inputs and outputs. The verifier V_θ receives the task, the primitive action (function call with inputs), a natural-language explanation of the step from the policy's in-line comments at (or before) the action, current observation, and the primitives already executed in this run (Equation (5) in Appendix A.5), and it either passes or blocks the primitive. On a block, the particular application setting may choose among handing the task to a stateful agent to continue, running NSPI, or escalating to a human.

Decision rule. We prompt a VLM to answer with one token, yes if the action should run now and no otherwise, following a fixed rubric (Appendix A.5). The verifier blocks when p_{no}, the renormalized probability mass of the no answers among the 20 most likely tokens, reaches a threshold θ set on a calibration set (Equation (6) in Appendix A.5). Using p_{no} instead of the greedy answer turns the trade-off between wrong blocks and wrong passes into a single parameter θ.

4 EXPERIMENTS AND RESULTS

4.1 EXPERIMENTAL SETUP

Benchmarks We evaluate on OSWorld-Verified (Xie et al., 2024) and ScienceBoard (Sun et al., 2025). We use 361 OSWorld tasks, excluding Google Drive tasks that require unavailable credentials, and 169 ScienceBoard tasks spanning six scientific desktop applications. Further evaluation settings are detailed in Appendix B.1. We also explore longer workflows on OSWorld-V2 (Yuan et al., 2026) as a separate extension (Section 4.5).

Compared methods Using GPT-5.6 Terra (OpenAI, 2026) and Claude Opus 5 (Anthropic, 2026) as backbone models, we compare our method with three baselines. The **base agent** plans each action from the current observation and solves each task anew on every run, without retaining experience across runs. **Batch2Code** is a conversion-only variant of NSPI that uses the same code-policy without iterative refinement. It constructs the policy from multiple base-agent trajectories, giving it more initial trajectory evidence than our single-trajectory initialization. **ASI** (Wang et al., 2025b) acquires reusable programmatic skills while retaining an agent to select and execute them. For reuse across parameter and initial-state variations, we compare with **AutoRPA** (Chen et al., 2026), adapted to learn reusable computer-use workflows from natural-language requests and execution experience, without benchmark-supplied task templates, parameter schemas, or oracle feedback. Further details on experiment setups and configurations are included in Appendix B.1.

Workflow variation We evaluate whether a learned policy transfers to a held-out instance of the same workflow. We constructed the workflow variations of the subset of OSWorld tasks. Parameter variants change request arguments, while initial-state variants preserve the request but alter the starting environment. We study 40 parameter and 35 initial-state families with up to three admitted variants. NSPI and AutoRPA both use GPT-5.6 Terra and a shared domain setup, without access to the benchmark evaluator. They process the same variants sequentially, refining a single policy as new instances expose task-specific assumptions. We then freeze the policy and evaluate Pass^3 on the held-out original task without further adaptation. We include implementation details in Appendix B.2.

Metrics We report Pass^k for $k \in \{1, 2, 3\}$, estimated from three evaluation trials per task as defined in Section 2.1; Pass^3 measures success in all three trials. We average over the full task or family set, retaining construction failures in the denominator. We report per-execution cost and wall-clock latency separately from one-time construction cost and time, and analyze their amortization over repeated use (Section 4.4).

4.2 Reliability across repeated executions

Table 1 shows a clear reliability gap under repeated execution. Averaged across the four benchmark–model settings, NSPI drops only 3.4 points from Pass^1 to Pass^3, compared with 12.4 points for the base agent, and achieves the highest Pass^3 in every setting, by 8.1 points on average over the next-best method. Its Pass^1 remains comparable to the base agent, so the flatter curve is not obtained by starting from a weaker policy. NSPI can also bootstrap from unsuccessful demonstrations. Across 356 tasks for which the base agent failed all three seed attempts, the final policy succeeds in Pass^3 on 62 tasks, including 32 of 89 on ScienceBoard with Opus (Figure 4). This recovery requires more refinement, with failed seeds incurring 18–31% higher healing cost and 17–38% longer healing time than successful seeds. We provide the full breakdown in Appendix C.

Model	Method	SR (%) Pass^1	SR (%) Pass^3	Cost ↓
ScienceBoard (169 tasks)				
GPT-5.6 Terra	Base agent	27.0	13.6	15×
	Batch2Code	15.4	11.8	1.1×
	ASI	24.3	11.8	16×
	NSPI (ours)	**29.2**	**26.0**	**0.00536**
Claude Opus 5	Base agent	38.7	29.6	65×
	Batch2Code	12.8	9.5	**0.9**×
	ASI	39.8	26.0	68×
	NSPI (ours)	**48.7**	**42.6**	0.00790
OSWorld (361 tasks)				
GPT-5.6 Terra	Base agent	**55.9**	34.6	67×
	Batch2Code	51.5	47.1	**0.9**×
	ASI	53.1	31.9	62×
	NSPI (ours)	53.6	**50.4**	0.00716
Claude Opus 5	Base agent	**74.1**	68.4	217×
	Batch2Code	46.7	45.2	**0.6**×
	ASI	73.8	67.6	381×
	NSPI (ours)	72.9	**72.0**	0.00965

Table 1: Reliability across repeated executions on ScienceBoard and OSWorld. Per-run cost is reported in USD for NSPI and as relative multiples (×) of NSPI for other methods.

The comparison with Batch2Code helps distinguish reliable execution from simply making execution more deterministic. Batch2Code uses the same executable policy representation without iterative refinement, and its Pass^k curve is similarly flat, but its Pass^3 remains 3.3–33.1 points below NSPI. Code therefore helps preserve a procedure across runs, while refinement makes it consistently correct. ASI, which instead retains an agent to select and execute learned programmatic skills, shows a decline with repeated execution closer to the base agent. Figure 3 illustrates the same trade-off on ScienceBoard with GPT-5.6 Terra. NSPI maintains its success rate as k increases while moving execution to a much lower-cost regime. Across all four settings, the resulting policies reduce per-run cost by 15-217x and run 3.4–5.1× faster than the corresponding base agents.

4.3 Policy generalization across workflow instances

We evaluate reuse across workflow instances on OSWorld with GPT-5.6 Terra, refining one policy on generated variants and evaluating the frozen policy on the held-out original task. As shown in Table 2, NSPI reaches 55.0% Pass^3 on parameter variations and 60.0% on initial-state variations, compared with 37.5% and 51.4% for AutoRPA. Both methods degrade a little from Pass^1 to Pass^3, suggesting that the dominant challenge is whether a learned policy transfers to a workflow instance at

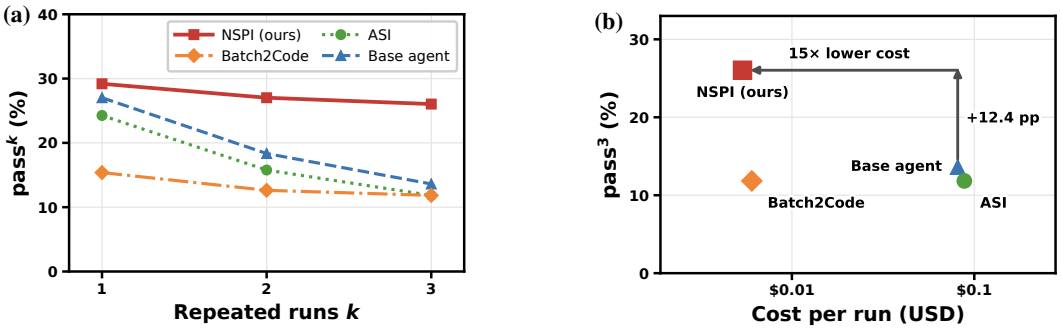


Figure 3: (a) Passˆk as k increases. (b) Passˆ3 versus per-run execution cost. Policy-construction cost is analyzed separately in Section 4.4.

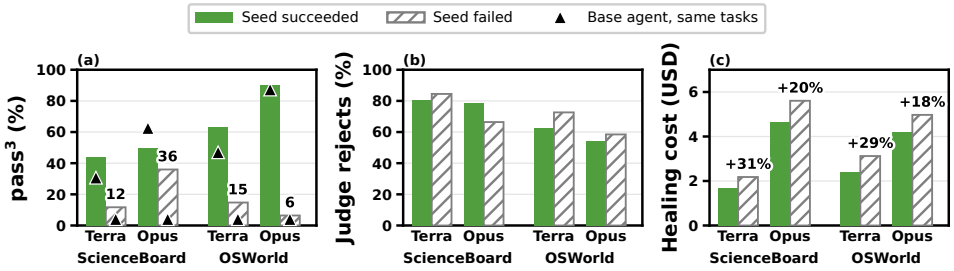


Figure 4: Final policies by seed outcome. (a) Passˆ3, with base-agent performance on the same tasks. (b) Judge rejection rate during policy iteration. (c) One-time healing cost with the relative increase for failed seeds.

all, rather than stochastic failure once it does. NSPI's advantage comes primarily from transferring successfully to more workflow instances.

Figure 5 examines which policy structures are associated with successful transfer. For parameterized workflows, policies that read changing values from the request transfer much more often than policies that retain training values, supporting the need to separate stable workflow structure from instance-specific inputs. For initial-state variation, policies that modify application files or use command-line interfaces transfer more often than GUI-only policies for both methods, consistent with being less dependent on the precise starting GUI state. AutoRPA relies on these script-based paths more heavily, yet NSPI performs better within both execution styles, so its advantage is not simply due to bypassing the GUI. Overall, successful reuse appears to depend on preserving the procedure while rebinding the parts that vary with the current request or environment.

4.4 Efficient repeated execution and cost amortization

We analyze the economics of neuro-symbolic policies using Eq. (3). Table 3 reports the resulting costs. Across different configurations, neuro-symbolic policy's per-execution cost ratio c_μ/c_π ranges from roughly 15× to roughly 217×, and policies recover their construction cost between 3.1 and 26.8 executions. The break-even points in wall-clock time range between 8.6 and 16.9 executions.

The results also indicate that the benefit of NSPI scales with task complexity. Moving from Science-Board to OSWorld increases the agent's per-run dollar cost by four to six times, while construction cost increases by only 14% to 52%. The break-even point drops nearly 75% in dollar value. The choice of a base model has a similar effect in dollar value. Claude Opus 5 incurs two to three times the construction cost of GPT-5.6 Terra, but its per-run agent cost is four to six times higher, so in dollars it breaks even roughly 2 to 2.4 times sooner. In wall-clock time, the two models are much closer. We note that these comparisons span only two benchmarks and two base models, so task complexity is partially confounded with other differences between benchmarks.

Setting	Method	Success rate (%) Pass^1	Pass^2	Pass^3
Parameters (40)	AutoRPA	37.5	37.5	37.5
	NSPI (ours)	**60.0**	**57.5**	**55.0**
Initial states (35)	AutoRPA	55.2	52.4	51.4
	NSPI (ours)	**61.0**	**60.0**	**60.0**

Table 2: Policy reuse across parameter and initial-state variations from OSWorld.

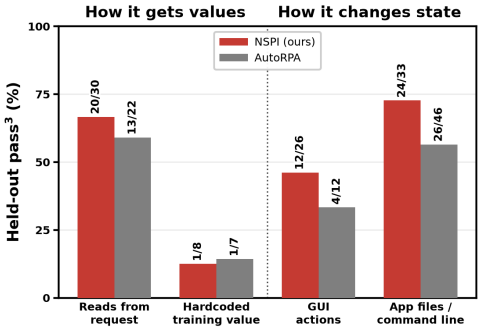


Figure 5: Policy reuse across parameter and initial-state variations. Labels show successful/total applicable policies. Non-applicable policies are omitted.

Table 3: Amortized cost of NSPI in the terms of Equation (3), means over tasks.

Benchmark	Base model	Cost (USD) C_π	c_μ	c_π	n^*	Wall-clock time C_π (min)	c_μ (s)	c_π (s)	n^*
ScienceBoard	GPT-5.6 Terra	2.03	0.081	0.0054	26.8	40	183	40	16.9
	Claude Opus 5	5.68	0.52	0.0079	11.2	49	251	64	15.8
OSWorld v1	GPT-5.6 Terra	3.09	0.48	0.0072	6.5	33	286	56	8.6
	Claude Opus 5	6.45	2.1	0.0097	3.1	43	353	102	10.3

4.5 Extension to long workflows on OSWorld-V2

We stress-tested NSPI on OSWorld-V2 (Yuan et al., 2026), where tasks take skilled humans more than an hour on average and cost frontier-model agents up to $76 and 50 minutes per task. Table 4 shows that the learned policies are more consistent than the base agent but less successful. On strict success, NSPI keeps 96% of its Pass^1 at k=3, whereas the base agent keeps only 80%. However, NSPI trails the base agent at k=3 by 5.6 points in strict success and by 18.5 points in partial credit. The gap in performance arises from a loss of information in converting a long agent trajectory to initial policy: agent trajectories averaged 158 steps and scripts averaged 970 lines, leading to mistakes when the script recomputes every value the agent read from the screen. Implementation details are available in Section B.5.

Method	Pass^1	Pass^2	Pass^3
Partial credit			
Base agent	**72.79**	**69.10**	**63.09**
NSPI	45.37	44.83	44.60
Strict success			
Base agent	**28.04**	**25.23**	**22.43**
NSPI	17.45	17.13	16.82

Table 4: Pass^k (%) on OSWorld-V2

4.6 Pre-action verification under environment variation

We evaluate the pre-action verifier (Section 3.2.2) with the injected intent comments and rubric additions of Appendix B.4 at $\theta = 0.78$. The *wrong-block rate* (WBR) is the fraction of rollouts of correct policies in which the verifier incorrectly blocks at least one primitive The *wrong-pass rate* (WPR) is the fraction of primitives that should be blocked but that the verifier passes. We simulate an unexpected environment change by running the correct policies of original OSWorld tasks but starting from the perturbed initial states of the generalization experiment (Section 4.3), which yields 24 primitives that should be blocked. Figure 8 shows that Qwen3.6-35B-A3B wrongly blocks at most 8.4% of correct rollouts in every setting, while Qwen3.8-27B wrongly blocks 9.4–25.0%. The trade-off reverses on the perturbed initial states: Qwen3.6-35B-A3B has WPR 70.8% (17 of 24 primitives passed), while Qwen3.8-27B has WPR 25.0% (6 of 24). Median verifier latency is $1.5\text{s} \pm 0.5$ (p95 2.3s) for Qwen3.6-35B-A3B and $1.6\text{s} \pm 1.2$ (p95 3.7s) for Qwen3.8-27B.

4.7 JUDGE ANALYSIS

Table 5 compares our judges with the benchmark evaluator on each task's final policy, where a policy is reliable only if it succeeds in all three runs. The judges reject 69.3–94.4% of unreliable policies but accept only 11.4–40.1% of reliable ones. They are therefore conservative, which increases construction cost without hurting reliability. A better-calibrated judge could lower this cost and potentially improve performance, which we leave to future work.

We also annotate 19 OSWorld policies rejected by J_{task} and compare both judges with human labels (Section B.3). Table 6 shows J_{task} agrees closely with human annotators, and so does J_{loc} on divergence localization when the evidence is visible on screen. We discuss four limitations of the judges in the same appendix.

Benchmark	Models	Prec.	Rec.	Spec.
ScienceBoard	Terra	41.7	11.4	94.4
	Opus 5	69.6	22.2	92.7
OSWorld	Terra	76.8	40.1	87.6
	Opus 5	76.7	35.3	69.3

Table 5: Judge vs. benchmark evaluator on final policies,

Judge output	Human Agreement
J_{task} verdict	84% (16/19)
t^*, visible failures	90% (9/10)
t^*, invisible failures	0% (0/6)

Table 6: Agreement of the two judges with human labels on 19 OSWorld tasks.

5 RELATED WORK

Prior methods on learning skills from experience enable an agent to accumulate a library of reusable APIs, represented either in natural language (Wang et al., 2025c; Zheng et al., 2025) or as short programs (Meng et al., 2025; Ning et al., 2026; Wang et al., 2023; 2025b). We take this approach to the limit by aiming for a single robust code policy that solves each equivalence set of parametric tasks in full. While the neural agent is the primary driver that invokes skills in prior work, our neuro-symbolic policy is primary while the agent is an expensive fallback.

Programs with perceptual inputs have long served as policies, from programmatic reinforcement learning (Verma et al., 2018) to LLM-generated robot code (Liang et al., 2023; Santos et al., 2026). The LLM-generated policies were produced from human natural language instructions and few-shot examples, instead of from a strong agent exploration as in our work. They were also static, whereas our neuro-symbolic iterative refinement method allows the policy to improve over time based on execution feedback and continual agent exploration.

The most similar prior work is AutoRPA (Chen et al., 2026), which converts ReAct-style agent exploration into replayable RPA scripts with iterative refinement, demonstrating substantial cost reduction while maintaining baseline pass@1 success rate. Our work differs in that our method explicitly optimizes Passˆk success rate and realistic intra-task generalization to different task parameters and initial states, along with a precondition check for practical deployment.

6 CONCLUSION

We introduced neuro-symbolic computer use as a way to turn execution experience into reusable policies for recurring workflows, preserving stable procedure in code while leaving state-dependent decisions to neural primitives. Neuro-Symbolic Policy Iteration learns these policies from agent trajectories through execution, diagnosis, exploration, and revision, and further refines them across task-parameter and initial-state variations. Across OSWorld and ScienceBoard, the resulting policies improve reliability under repeated execution while substantially reducing per-run cost and latency, and they transfer more successfully than AutoRPA to held-out instances of the same workflow. Results on longer-horizon tasks and our judge and verifier analyses also expose remaining bottlenecks in trajectory-to-policy conversion and policy evaluation, suggesting that better acquisition and verification could further extend this approach to more complex recurring workflows.

AI USE STATEMENT

In this work, we used generative AI tools for generating workflow variations used in our experiments and to assist on implementation of parts of the experiments. We have not used generative AI tools for developing 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, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, and interpret results. Additionally, we used generative AI tools for polishing writing and creating scientific figures. We have reviewed all AI-assisted work. We manually reviewed the generated workflow variants to finalize on the subset used for the experiments. LLM-generated code was verified and tested for correctness by 3 authors, and all visualizations were cross-checked with original results. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Wang. Agent s: An open agentic framework that uses computers like a human. In *International Conference on Learning Representations*, volume 2025, pp. 22924–22946, 2025a.

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025b.

Anthropic. Claude opus 5 system card. https://www.anthropic.com/news/claude-opus-5, 2026.

Minghao Chen, Xinyi Hu, Zhou Yu, and Yufei Yin. AutoRPA: Efficient GUI automation through LLM-driven code synthesis from interactions. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=RbGLc4sk9s.

Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

Gonzalo Gonzalez-Pumariega, Saaket Agashe, Jiachen Yang, Ang Li, and Xin Eric Wang. On the reliability of computer use agents. *arXiv preprint arXiv:2604.17849*, 2026a.

Gonzalo Gonzalez-Pumariega, Vincent Tu, Chih-Lun Lee, Jiachen Yang, Ang Li, and Xin Eric Wang. Scaling agents for computer use, 2026b. URL https://arxiv.org/abs/2510.02250.

Ronald A Howard. Dynamic programming and markov processes. 1960.

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.

Yuan Meng, Zhenguo Sun, Max Fest, Xukun Li, Zhenshan Bing, and Alois Knoll. Growing with your embodied agent: A human-in-the-loop lifelong code generation framework for long-horizon manipulation skills. *arXiv preprint arXiv:2509.18597*, 2025.

Microsoft. Playwright: Fast and reliable end-to-end testing for modern web apps. https://github.com/microsoft/playwright, 2020. Open-source library.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

OpenAI. GPT-5 release notes. https://openai.com/index/gpt-5-6/, 2026. Accessed: 2026-07-25.

Tejal Patwardhan, Rachel Dias, Elizabeth Proehl, Grace Kim, Michele Wang, Olivia Watkins, Simon Fishman, Marwan Aljubeh, Phoebe Thacker, Laurance Fauconnet, et al. Gdpval: Evaluating ai model performance on real-world economically valuable tasks. In *International Conference on Learning Representations*, volume 2026, pp. 24005–24040, 2026.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents, 2025. URL https://arxiv.org/abs/2501.12326.

Vitor Gaboardi dos Santos, Ibrahim Khadraoui, Ibrahim Farhat, Hamza Yous, Samy Teffahi, and Hakim Hacid. Alrm: Agentic llm for robotic manipulation. *arXiv preprint arXiv:2601.19510*, 2026.

Simular AI. Simulang: A scripting language for desktop and web automation. https://github.com/simular-ai/simulang, 2026. Open-source repository, accessed 2026.

Qiushi Sun, Zhoumianze Liu, Chang Ma, Zichen Ding, Fangzhi Xu, Zhangyue Yin, Haiteng Zhao, Zhenyu Wu, Kanzhi Cheng, Zhaoyang Liu, et al. Scienceboard: Evaluating multimodal autonomous agents in realistic scientific workflows. *arXiv preprint arXiv:2505.19897*, 2025.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2nd edition, 2018.

Al Sweigart. PyAutoGUI: Cross-platform gui automation python module for human beings. https://github.com/asweigart/pyautogui, 2014. Open-source library.

Venus Team, Changlong Gao, Zhangxuan Gu, Yulin Liu, Xinyu Qiu, Shuheng Shen, Yue Wen, Tianyu Xia, Zhenyu Xu, Zhengwen Zeng, et al. Ui-venus-1.5 technical report. *arXiv preprint arXiv:2602.09082*, 2026.

Wil MP Van der Aalst, Martin Bichler, and Armin Heinzl. Robotic process automation. *Business & information systems engineering*, 60(4):269–272, 2018.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International conference on machine learning*, pp. 5045–5054. PMLR, 2018.

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.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, et al. Opencua: Open foundations for computer-use agents, 2025a. URL https://arxiv.org/abs/2508.09123.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing programmatic skills for agentic tasks. In *Second Conference on Language Modeling*, 2025b. URL https://openreview.net/forum?id=lsAY6fWsog.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Forty-second International Conference on Machine Learning*, 2025c. URL https://openreview.net/forum?id=NTAhi2JEEE.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, et al. Osworld2. 0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026.

Table 7: Primitive action space of policies on ScienceBoard. Each primitive is a method of an `agent` object that the policy calls, e.g., `agent.click("File menu")`. Neural primitives call a model on the current observation o_t: the grounded actions use UI-Venus-1.5-30B-A3B (Team et al., 2026) with a consensus over three parallel calls, and the condition check uses Gemini 3 Flash. A check mark under *Mut.* marks a primitive that changes the state of the computer; these count toward the per-task step budget and are gated by the pre-action verifier.

Primitive	**Mut.**	**Behavior**
Neural: grounded GUI action $G(\ell, o_t)$		
`click(`ℓ`, clicks, button, hold_keys)`	✓	Grounds the element description ℓ to a screen coordinate and clicks it; `clicks` sets double or triple clicks and `hold_keys` holds modifiers.
`type(`ℓ`, text, enter, overwrite)`	✓	Clicks the element grounded from ℓ if given, clears it if `overwrite`, types `text`, and presses Enter if `enter`.
`scroll(`ℓ`, amount)`	✓	Moves the pointer to the element grounded from ℓ and scrolls by `amount`.
`drag_and_drop(`ℓ_{start}`,` ℓ_{end}`, hold_keys)`	✓	Grounds both descriptions and drags from the first element to the second.
Neural: condition check $Q(\phi, o_t)$		
`state_satisfies(`ϕ`)` $\rightarrow$ `bool`		Returns whether the current screenshot satisfies the natural-language condition ϕ; the plan can branch on the result.
Symbolic: keyboard and applications		
`hotkey(keys)`	✓	Presses a key combination, e.g., `["ctrl", "s"]`.
`hold_and_press(hold, press)`	✓	Holds the keys in `hold` while pressing the keys in `press`.
`open(path)`	✓	Opens a file with its default application, or launches an application by name.
Symbolic: programmatic		
`_exec_python(code)` $\rightarrow$ `str`	✓	Runs Python on the computer and returns its standard output.
`_exec_bash(cmd)` $\rightarrow$ `str`	✓	Runs a shell command on the computer and returns its standard output.
Symbolic: control		
`wait(seconds)`		Waits for up to 30 seconds, e.g., for a dialog to open.
`done()`		Ends the plan and reports the task as complete.
`answer(x)`		Ends the plan and submits `x`, which the evaluator matches exactly, for question-answering tasks.
`fail()`		Ends the plan and reports the task as infeasible.

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*, 2025.

A Method details

A.1 Primitive action space

We instantiate policies in Simulang (Simular AI, 2026), a scripting language for desktop and web automation in the style of PyAutoGUI (Sweigart, 2014) and Playwright (Microsoft, 2020) that also provides the neural primitives above. Table 7 lists the primitives that the converter and the healer can use on ScienceBoard. The healer can combine them with Python control flow, e.g., a loop that retries a click or a branch on `state_satisfies`.

Algorithm 1 Rollout diagnosis with the dual judge (policy evaluation)

Require: rollout $\tau = (o_0, a_0, \ldots, o_{|\tau|})$; task g; policy π; task-completion judge J_{task}; observer J_{obs}; localizer J_{loc}; applicability gate APPLICABLE; aggregator AGGREGATE

procedure DIAGNOSE(τ, g, π)
  $(v_{\text{task}}, e_{\text{task}}) \leftarrow (\text{skipped}, \emptyset)$
  if APPLICABLE(g) **then** ▷ is success checkable from persistent state?
    $(v_{\text{task}}, e_{\text{task}}) \leftarrow J_{\text{task}}(g)$ ▷ read-only probes of the end state, in parallel with the step-level judge
  end if
  for $t = 0, 1, \ldots, |\tau| - 1$ **do in parallel** ▷ blind to g, π, and step intent
    $\delta_t \leftarrow J_{\text{obs}}(o_t, o_{t+1})$
  end for
  $(v_{\text{loc}}, t^*, m) \leftarrow J_{\text{loc}}(g, \pi, \delta_{0:|\tau|-1})$ ▷ verdict, first divergent step, failure mode
  $v \leftarrow$ success **if** $v_{\text{loc}} =$ success **and** $v_{\text{task}} \in \{\text{success}, \text{skipped}\}$ **else** failure ▷ abstain counts as failure
  return $\big(v, \text{AGGREGATE}(v_{\text{task}}, e_{\text{task}}, v_{\text{loc}}, t^*, m)\big)$ ▷ verdict and diagnostic d
end procedure

A.2 INITIALIZATION

The base agent μ runs on g in exploration mode, in which actions may be arbitrary code, and we record its trajectory τ_μ. The converter Φ_{conv} first extracts the discrete actions of τ_μ into a single contiguous script (Section 4.1). It then replaces pixel coordinates, which were recorded on a different machine and are unreliable, with natural-language descriptions of the target elements that the runtime grounds at execution time, and it replaces values specific to g with task parameters. On ScienceBoard, this second step is one LLM call that receives the task, the extracted script, and a screenshot and accessibility tree of the initial state. On OSWorld, the base agent trajectory was based on Agent-S3 (Gonzalez-Pumariega et al., 2026b) which already encodes the trajectory in semantic-level rather than pixel-level, then calls an LLM for the final conversion.Conditioning Φ_{conv} on g lets it drop incidental actions in τ_μ, such as backtracking, exploratory clicks, and redundant verifications, so that they are not transcribed into the code policy that will be refined.

A.3 ITERATION

The rollouts within an iteration run in a random order, and the iteration stops at the first rollout that fails the judge, so the agent continuation runs at most once per iteration. Every code rollout and every agent continuation is capped at the task's step budget, i.e., the maximum number of state-mutating actions, which matches the budget of the baseline agent; the localizer and the healer are told the budget so that their suggestions fit within it. The agent continuation can use GUI actions and run Python or shell commands on the machine, and it receives the output of those commands in later turns. The continuation is not judged and does not change the verdict of the code rollout; it serves only as experience for the healer. A rollout that raises a code execution error (a syntax error, an undefined name, a failed import, etc.) produces no trajectory for the judges to grade, so we skip the judges and the continuation and give the traceback to the healer. The healer receives at most five screenshots of the failed rollout, sampled evenly over the trajectory, and it rewrites the whole code policy. The converter and the healer are instructed to prefer programmatic actions (shell commands, Python, and application command lines) over long sequences of GUI actions, which replaces neural primitives with symbolic ones and lowers the cost per execution c_π, although our method does not optimize cost directly.

A.4 DUAL JUDGE

Motivation for two judges. Evaluating a computer-use rollout is harder than reading an exit code, for two reasons. First, an agent's own report of what it did is often optimistic, e.g., it may report that it clicked a button while the screen shows that the click missed, or that it entered a value into a spreadsheet cell while it edited the wrong cell, so an evaluator that trusts the actor's narrative inherits its errors. Second, benchmark evaluation scripts such as those of OSWorld check the end state of

the environment (the contents of a saved spreadsheet, an exported image, the browser's bookmarks, etc.), which an evaluator that only watches the trajectory cannot see. For example, a LibreOffice Calc plan that types the correct formula but closes the window before the file is saved, a GIMP plan whose edit is never exported to the target file, and a Chrome plan whose bookmark fails to persist all look correct frame by frame, yet all fail the evaluation script. The task completion judge addresses the second problem, and the step-level judge addresses the first while also localizing the error for the healer. We ablate the two judges in Section 4.7.

Task completion judge. J_{task} is a VLM agent with read-only probes of the live machine: shell commands, file reads, directory listings, SQL queries over application databases, and screenshots. Mutating shell commands are rejected by a deny-list, and databases are opened read-only. J_{task} receives only the task description (and the submitted answer for question-answering tasks), and never another judge's verdict or the benchmark's verdict. It decomposes the goal into verifiable sub-conditions, probes the machine until it can settle each one (preferring filesystem and database probes over screenshots), and returns a verdict with supporting evidence within a budget of 30 tool calls and 300 seconds. It abstains (`cannot_verify`) when the state needed to check success is transient or depends on context that is not persisted, e.g., for "move the file back where it used to be", the original path no longer exists. J_{task} is not a ground-truth oracle, because its verdict is only as good as the probes it chooses to run; this is why an abstention falls back to the step-level judge.

Applicability gate. One VLM call on the task instruction, cached per task, decides whether success can be checked from persistent machine state. If not, J_{task} is skipped and the verdict comes from the step-level judge alone, which spends no probe budget on a question J_{task} could not answer.

Observers. For GUI primitives, the evidence for transition t is the pair of screenshots before and after the step, together with the difference between accessibility trees when available. For primitives that leave no visual trace, e.g., API calls, background shell commands, or functions executed inside the code runtime, the evidence is the return value, exit code, and captured output, and no screen change is expected. Each observer returns an objective description of what the step produced and an anomaly grade. Observers do not see the task, the code, or the stated intent of the step, because an observer told that the agent intended to save a file can no longer distinguish "the agent believes it saved" from "the file was saved", which is the distinction the evaluation script checks. The $|\tau|$ observer calls are independent and run in parallel.

Localizer. The localizer is one VLM call that receives the task, the executed code, and the ordered observer outputs, but not the actor's narrative. It enumerates the sub-goals of the task and checks each against the observer evidence. It also screens for composition failures, i.e., sequences of individually plausible steps that together are incoherent or destructive, such as an action that is immediately undone, a target value that is overwritten, or a save followed by closing the application before the write completed. It returns the verdict, the first divergent step t^*, and a name for the failure mode, which the healer uses when rewriting the code policy.

Diagnostic. An LLM composes the outputs of both judges into the diagnostic d. d lists each judge's observations (the probe results of J_{task} and the observer descriptions) separately from its interpretations (verdicts, failure hypotheses, and suggested focus), and it tells the healer that observations are facts about the machine while verdicts can be wrong.

A.5 Pre-action verifier

Verifier and safety guardrails. A safety guardrail confirms that an irreversible action is intended, e.g., a final `click({concept: "Pay now"})`, and is invoked at a small number of known points. The pre-action verifier gates every state-mutating primitive, including reversible ones, against the implicit assumptions under which the line of code was written. A click on an email field is reversible, but if a cookie-consent banner is absorbing pointer events, the click is lost and every later step operates on an unchanged page. We therefore do not skip the check for reversible actions, since a downstream component that could undo a wrong execution may not exist.

Verifier and in-code state checks. A code policy may also contain its own state checks, e.g., a `stateSatisfies({condition})` line inserted where the planner expects a particular state. Three properties separate these from the verifier. First, in-code checks run only where the planner placed them, whereas the verifier runs before every gated primitive. Second, an in-code check carries

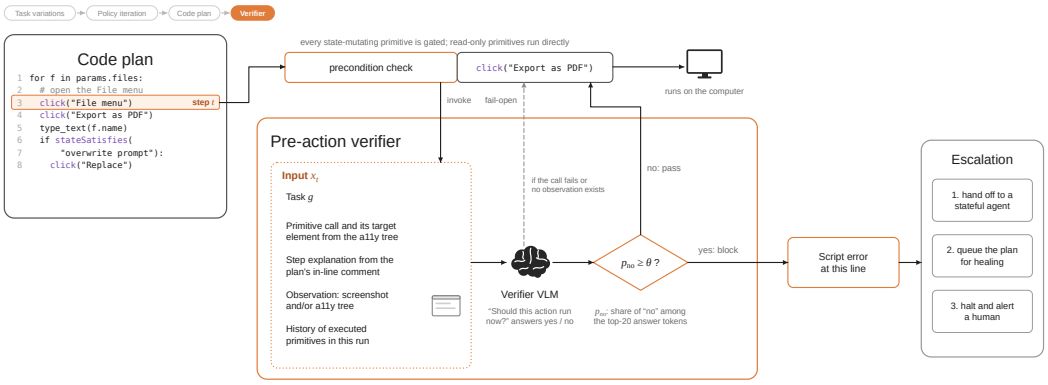


Figure 6: A learned binary-output pre-action verifier V runs before every state-mutating primitive during policy deployment. Reusing the plan's most recent stashed screenshot and accessibility tree, V emits top-20 answer-token logprobs; the yes/no log-mass is renormalized to p_{no} and the action is blocked iff $p_{\text{no}} \geq \theta$. A block raises a script-level error at the offending line, triggering the fallback hierarchy (agent hand-off, code repair, halt).

whatever condition string the planner wrote, which differs across plans and cannot be improved without rewriting the plan, whereas the verifier uses one fixed prompt that we version separately from any code policy. Third, the verifier exposes the threshold θ, whereas an in-code check returns a hard yes or no. We treat in-code checks as a planner affordance for control flow, and they do not replace the verifier.

Pre-action and post-action checks. Any error that a post-action check on a_t could detect, i.e., a primitive that returned successfully without its intended effect, is also detectable by the pre-action check on a_{t+1}, because the precondition of a_{t+1} was written against the intended post-state of a_t. A pre-action check therefore covers post-action checking up to a lag of one step, uses the same observation as the next action, and leads directly to the fallback, since the response to a failed check is to not execute the next primitive. The final primitive of a plan has no successor to catch its failed effect; the runtime reserves a post-action slot for an outcome check there, and end-of-rollout verification is provided by the task completion judge.

Gated primitives. We gate primitives that write to the state of the computer: pointer and keyboard input, form mutation, navigation, filesystem and clipboard writes, integration writes such as spreadsheet cell updates, and in-page script execution, whether called directly or as a method on an application or page handle. In the ScienceBoard and OSWorld runtime these are `click`, `double_click`, `type`, `scroll`, `hotkey`, `hold_and_press`, `open`, `drag_and_drop`, `set_cell_values`, and Python and shell execution. We gate reversible write primitives too, because an undo path may not exist (e.g., a click that triggers an analytics event), and because a skilled human operator would not execute a step against an obviously wrong state regardless of whether the step is reversible. We do not gate read-only primitives (screenshots, accessibility snapshots, reading page content or the clipboard), because they cannot corrupt external state and their outputs are the observations that the verifier consumes. We also do not gate primitives confined to the runtime (variable assignment, control flow, in-memory list operations), whose outcome is determined by the code and cannot disagree with the external world, or window and tab lifecycle operations, which typically run before any observation exists. When membership is ambiguous we gate the primitive, because a spurious check costs one verifier call that usually passes at a low wrong-block operating point. Writes issued through headless APIs, e.g., sending mail through a mail-service API, present no on-screen state to verify against and need a separate safeguard.

Inputs. Before executing a gated primitive at step t, the runtime builds the input

$$x_t = (g, c_t, e_t, o_t, h_t), \tag{5}$$

where c_t is the code of the primitive call; e_t is a natural-language explanation of the step taken from the in-line comments of the code policy; o_t is the current observation, as a screenshot, an accessibility tree, or both; and h_t lists the primitives already executed in this run. The verifier V_θ

maps x_t to a decision in $\{\text{pass}, \text{block}\}$. The explanation e_t is the trailing comment on the statement that contains the call, otherwise the comment block directly above it, otherwise the first line of the enclosing function's docstring, and otherwise a literal restatement of the call. A verifier note supplies instance-specific knowledge that the verifier cannot infer, e.g., a machine-specific path or the operating system; the note is presented as a fact about the environment and not as approval, so the rubric still applies. The history h_t is a compact list of the steps completed so far in the run, which lets the verifier distinguish a repeated action from its first occurrence without a full transcript.

Log-probability aggregation. We request the 20 most likely next tokens with their log-probabilities at the answer position, skipping any reasoning block that precedes the answer. Let $\ell_t(w)$ be the log-probability of token w at the answer position, restricted to the 20 most likely tokens, and let $\mathcal{W}_{\text{yes}}$ and $\mathcal{W}_{\text{no}}$ be the surface forms of each answer (e.g., `yes`, `Yes`, `Y`). The probability of blocking is the renormalized mass of the no forms, and the verifier blocks when it reaches a threshold θ that is set by a calibration set (see Calibration below):

$$p_{\text{no}}(x_t) = \frac{\sum_{w \in \mathcal{W}_{\text{no}}} \exp \ell_t(w)}{\sum_{w \in \mathcal{W}_{\text{yes}} \cup \mathcal{W}_{\text{no}}} \exp \ell_t(w)}, \qquad V_\theta(x_t) = \begin{cases} \text{block} & \text{if } p_{\text{no}}(x_t) \geq \theta, \\ \text{pass} & \text{otherwise.} \end{cases} \tag{6}$$

The yes forms are `yes`, `Yes`, `YES`, `y`, `Y` and their variants with a leading space, and the no forms are defined likewise. We compute the mass of each class with a log-sum-exp over its forms and renormalize over the two classes to obtain p_{no} (Equation (6)). If neither class appears among the 20 tokens, we use the greedy answer, and if the completion contains neither answer, the verifier fails open.

Calibration. A user who automates a repetitive task expects a working code policy to run without interruption, so a wrong block (stopping a correct primitive) costs more than a wrong pass (running an incorrect primitive), as long as safety guardrails are in place. We therefore set θ in a calibration phase before any block is enforced. We run the verifier in shadow mode, which computes and logs p_{no} for every gated primitive without enforcing blocks, on a calibration set of rollouts of final policies, and we label each rollout with its task outcome. We treat every primitive of a successful rollout as correct, so any block within a successful rollout is a wrong block. Let $\text{WBR}(\theta)$ be the fraction of successful rollouts in which at least one primitive has $p_{\text{no}} \geq \theta$, and $\text{DR}(\theta)$ the fraction of failed rollouts in which at least one primitive has $p_{\text{no}} \geq \theta$. Both rates are non-increasing in θ, so for a wrong-block budget ϵ chosen by the deployment, the threshold with the highest detection rate within the budget is the smallest feasible threshold on a grid Θ:

$$\theta^* = \min\{\theta \in \Theta : \text{WBR}(\theta) \leq \epsilon\}. \tag{7}$$

B Experiment details

B.1 Main experiment configuration

Methods. Both backbones, GPT-5.6 Terra and Claude Opus 5, are called through OpenRouter, and every method on a benchmark uses the same backbone in every LLM role except the judges, the `state_satisfies` checker, and the continuation summarizer (Table 8). The *base agent* is Agent-S3 (Gonzalez-Pumariega et al., 2026b) on OSWorld, with UI-Venus-1.5-30B (Team et al., 2026) as its grounding model, and ScienceBoard's own screenshot agent on ScienceBoard. *NSPI* starts from one base-agent trajectory per task (Section A.2) and runs policy iteration with $N{=}5$ iterations of up to $K{=}5$ rollouts. *Batch2Code* converts three base-agent trajectories per task into a policy with one call of the same converter and no refinement. *ASI* (Wang et al., 2025b) builds its skill library in one online pass over the tasks on top of the same base agent; its later trials replay the agent with the library frozen after that task. Code policies of NSPI and Batch2Code share the grounding model and run under the same step limits.

Scoring. Each method gets three fresh trials per task, from which we estimate Pass^k for $k \in \{1, 2, 3\}$ over the full task set. Cost is the provider-billed model cost plus the grounding model at \$3.25 per GPU-hour of call time. One-time construction cost covers conversion and policy iteration (NSPI), conversion (Batch2Code), or the online library-building pass (ASI); per-run cost covers one final trial.

Setting	ScienceBoard	OSWorld
Base agent: output tokens, step limit	8192; task step limit	4096; 100 steps
Code rollout step limit	task limit $\times 5$ (Terra), $\times 16$ (Opus)	100 primitive calls
Converter / healer / continuation tokens, Terra	2000 / 4096 / 1024	4096 / 8192 / 1024
Converter / healer / continuation tokens, Opus 5	4096 / 16384 / 2048	4096 / 16384 / 2048
Continuation turns	15	15
Observer / localizer (J_{loc})	Gemini 2.5 Flash / Gemini 2.5 Pro	
Task-completion judge J_{task}	Claude Sonnet 4.6 (30 tool calls, 300 s)	
Applicability gate / judge combination	Gemini 2.5 Pro / Gemini 2.5 Flash	
`state_satisfies`, continuation summarizer	Gemini 3 Flash	

Table 8: Main-experiment configuration. All judge roles use temperature 0.

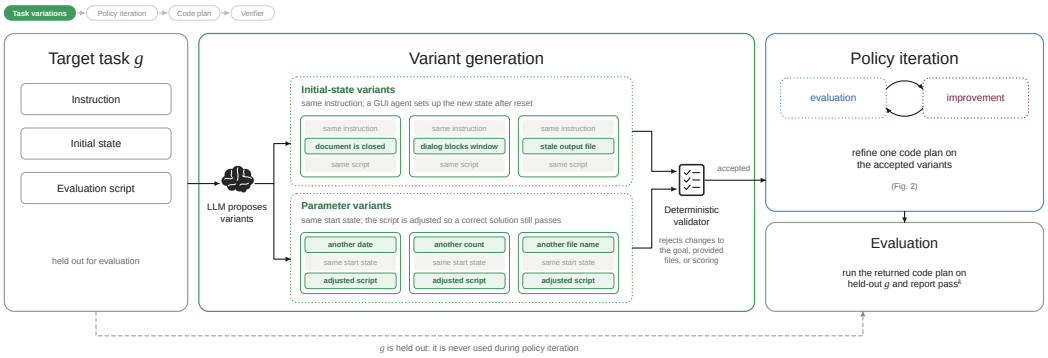


Figure 7: Given an original target task and environment configuration (initial state and evaluation script), we call an LLM to generate synthetic task variants that differ in the initial state and task parameters

B.2 Workflow Variation

B.2.1 Task variant generation

We generate the variants of each task with one LLM call followed by a deterministic validator (Figure 7). The LLM receives the full task specification, i.e., the instruction, the environment setup, and the evaluation script, together with notes that we extract from the evaluation script, e.g., whether it compares the output against a pre-rendered reference file or against inline constants. It first decides for each category whether the task admits variants at all, and then proposes up to three variants per admissible category. An initial-state variant keeps the instruction and adds a setup instruction that a general GUI agent executes before the policy runs, e.g., "Close the spreadsheet window entirely, leaving only the file manager open." The setup instruction may only change transient application state (open windows, selected tabs, dialogs, etc.) or pre-fill the region that the task writes to, so the files provided by the task setup and the original solution stay unchanged. A parameter variant rewrites the instruction with new argument values and rebinds the constants in the evaluation script to match. The validator rejects a parameter variant if its instruction is unchanged, if the rebound evaluation script uses a different scoring function, or if the script still compares a changed output against the original reference file, because a correct solution could never match that file. It marks variants that pass but carry a warning, e.g., an edited evaluation script on an initial-state variant, for human review. Because the LLM may decline a category, parameter variants are much rarer than initial-state variants: on OSWorld, 332 of 369 tasks pass a pre-filter that removes infeasible tasks, all 332 admit initial-state variants, and only 93 admit parameter variants, mainly because 161 of the other 239 evaluation scripts compare against reference files that we cannot regenerate. We manually review the generated variants and remove infeasible cases. We retain only families whose original task was solved at least once by the Terra or Opus 5 base agent, so failures measure transfer rather than task infeasibility. The resulting benchmark contains 40 parameter families and 35 initial-state families across ten OSWorld domains, generally with three variants per family.

Construction protocol. Both methods use GPT-5.6 Terra for all construction roles and the same grounding model with UI-Venus-1.5-30B Team et al. (2026). Variants are processed sequentially while carrying a single policy across the family. Construction starts from each method's own exploration and never exposes the original task. Only model-judge feedback is available during construction and benchmark evaluators are reserved for final evaluation. NSPI performs up to three refinement iterations with up to three judged rollouts per variant. J_{task} determines rollout success, while J_{loc} provides failure localization for healing. This gated setup was adapted due to the sequential aspect of the experiment meant that lenient judges are more appropriate over strict judges due to the chained dependency.

AutoRPA adaptation. We adapt the AutoRPA reference implementation (Chen et al., 2026) to computer-use environments while preserving its exploration, reflection, breakpoint continuation, newest-first verification, and rollback logic. We remove inputs unavailable from a normal user request—the task template, named parameter schema, and benchmark reward. Both methods instead read the current request at execution time, use J_{task} in place of reward, and share the same visual grounding model.

Evaluation. After construction on all variants, the policy is frozen and evaluated three times from fresh environments on the original task. If c of the three runs succeed, the family contributes $\binom{c}{k}/\binom{3}{k}$ to Passˆk; results are averaged across families, with construction failures assigned zero.

B.3 JUDGE ANALYSIS DETAILS

Annotation. We annotate 19 OSWorld tasks whose initial code policy failed according to J_{task}, one rollout per task (Table 9). We leave out tasks that fail because a Python library is missing in the guest; these are the most common failures, and the step-level judge cannot see them (see the first limitation). One annotator labels each rollout. The annotator sees the task, the benchmark evaluator, the code policy, one screenshot per step, and the output of every shell command and script. For each rollout, the annotator records five labels: 1) whether the rollout really failed; 2) the divergence step; 3) the first visible divergence, i.e., the first step at which a person who sees only the screenshots can tell that the rollout went wrong, or none if the failure never shows on screen; 4) the type of failure; 5) the cause, in one or two sentences. The annotator then sees the output of the step-level judge and rates it: whether t^* is correct, adjacent (within the same sub-goal), or wrong; whether the narrative names the real cause; whether the suggested focus would steer the healer to the right fix; and whether the observer described the divergence step correctly.

Ground truth. The annotator judges success against the task instruction and uses the benchmark evaluator as a reference. The two disagree on one rollout, in which the plan wrote network speeds from a command-line tool instead of copying them from the web page that the task names. The evaluator only checks that the file names the three metrics, so it passes the rollout, but the annotator counts it as a failure.

Task completion judge. J_{task} is correct on 16 of the 19 tasks. The annotator judges the other 3 tasks as successful, so J_{task} gave false alarms on them. In two of the false alarms, J_{task} read outdated values from saved files; in the third, it read a default value written by a web tool as a violation of the instruction.

Step-level judge. We compare t^* with the first visible divergence on the 16 confirmed failures. In 6 of them, the failure happens only inside a script and is never visible to the step-level judge; t^* is wrong in all 6. In the other 10, the failure becomes visible on screen at some step. Against the first visible divergence, t^* is correct in 7 of these 10, adjacent in 2, and wrong in 1. The narrative names the real cause in 4 of the 10, names it in part in 4, and is wrong in 2. The suggested focus would steer the healer to the right fix in 7 of the 10.

Errors in visible failures. Of the 10 visible failures, t^* misses the first visible divergence in 3. In one, the observer reported no change although a menu failed to open, so t^* points four steps later, where the menu closes. In another, the plan fell one step behind; the judge named the step next to the divergence in its suspect chain but reported a later step as t^*. In the third, the judge matched a condition check to a click in the code, because condition checks write no step-log entry, and placed t^* three steps before the failure shows on screen.

Limitations. We find four limitations.

First, the step-level judge receives no return values from the primitives. It sees the code policy and the screenshots, but not the output of shell commands and scripts or the answers of condition checks, and condition checks appear as screen transitions with no action. This limitation also hides the most common failure, a missing library, because the error appears only in the script output; our sample leaves these failures out, so the counts above are more favorable to the step-level judge.

Second, when a click misses or registers late, the code policy falls one step behind, and each later action hits the wrong target. The step-level judge then places t^* at the later symptom instead of the step that the healer needs to fix. For instance, a short wait causes the next click to fire before the target has loaded.

Third, neither judge recognizes an infeasible task. Both infeasible tasks in the sample ask for something that does not exist: a "Blue" theme in GIMP and a photo in a folder that has none. Both judges suggest how to complete the task instead of reporting that it is infeasible.

Fourth, J_{task} trusts saved files and the output of its own tools more than the running application. Saved files can lag the state of the application, and tool output can contain text that is not in the file.

Table 9: The 19 annotated rollouts. Step: true divergence step. Visible: first visible divergence (–, never). t^* rating against the true step and the first visible divergence: C correct, A adjacent, W wrong.

ID	App	Step	Visible	t^*	Rating	Cause
R01	Chrome	27	27	27	C / C	Site is sold out; plan should report the task as infeasible
R02	Chrome	3	22	22	W / C	`date -d monday` returns today, not next Monday
R03	GIMP	9	9	13	A / A	Click on Mode does not open the menu; next click reuses its point
R04	GIMP	14	18	15	A / W	Condition check returns empty; Export skipped; Replace dialog left open
R05	GIMP	6	6	9	A / A	Wait too short; click misses; plan falls one step behind
R06	GIMP	2	5	5	W / C	Infeasible: plan writes a "Blue" theme that does not exist
R07	Calc	7	9	9	A / C	Script turns numbers into text; formulas show `#VALUE!`
R08	Impress	8	8	8	C / C	Click reopens the background list; Pattern set; hex typed into slide
R09	Multi	2	–	0	W / –	Regex bug; script appends headings instead of filling placeholders
R10	Multi	9	–	15	W / –	Values from a command-line tool, not the web page
R11	Multi	5	–	9	W / –	Invoice month taken from the email send date
R12	Multi	–	–	8	–	Success; J_{task} reads a tool default as a violation
R13	Multi	5	5	5	C / C	Drag and drop does not move the sheet tabs
R14	Multi	2	–	0	W / –	Conversion to xlsx fails after the plan kills the open Calc
R15	Multi	4	–	1	W / –	Unlabeled column; script raises an error and writes nothing
R16	OS	–	–	4	–	Success; J_{task} reads an old dconf value
R17	Thunderbird	–	–	12	–	Success; J_{task} reads the mail file before the star is saved
R18	VS Code	2	7	7	W / C	Infeasible: plan uses a server screenshot as the photo
R23	Impress	4	–	8	W / –	Script underlines the title as well as the body

B.4 Precondition checks

The verifier receives a natural-language explanation of each primitive alongside its code. At deployment, the runtime takes this explanation from the comments of the code policy: a trailing comment on the statement that contains the call, otherwise the block of comment lines directly above that statement, otherwise the first line of the docstring of the enclosing function, and otherwise a literal restatement of the call (e.g., "press ctrl+q", "run a shell command on the VM"). The final code policies produced by policy iteration carry few such comments: on OSWorld with the Terra converter, 99.9% of gated primitives fall back to the literal restatement. A literal restatement gives the verifier no reason for steps that are correct within

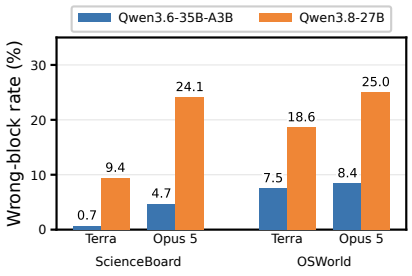


Figure 8: Pre-action verifier error rates at $\theta = 0.78$. Rollout-level wrong-block rate on correct policies, by environment, base model used for initial trajectory seed, and verifier model.

the plan's strategy but look destructive in isolation, such as closing an application so that a later step can edit its file on disk. We therefore inject one comment per gated primitive that states the intent of the step within the overall code policy, using the following procedure.

Targets. We parse each plan and find every call to a gated primitive (`click`, `type`, `hotkey`, `open`, `scroll`, `drag_and_drop`, `set_cell_values`, shell and Python execution, etc.). We group calls by the statement that the runtime uses to look up the explanation, and we leave a statement unchanged if it already has a trailing comment or a comment on the line directly above it, so that existing comments written during refinement are preserved.

Intent generation. For each plan we make one call to Gemini 3.8 Flash (low thinking level) with the task instruction, the plan with line numbers, and the target statements, and the model returns one comment per target as JSON. Each comment starts with the action and its target in the words of the code, then explains why the plan takes the step at that point in its strategy. For a step that could look wrong out of context, e.g., closing or killing an application, saving, or running shell code instead of using the GUI, the comment says why the step is needed, and for a close, kill, or save it names the later step that depends on it. A step inside a retry branch is described as retrying an attempt that the preceding check found did not take effect. The comment must not rename or correct the target, claim facts the plan does not establish (e.g., that a file was already saved), or merely restate the call, and it fits on one line of at most 220 characters.

Insertion and verification. The model writes only comments, and a script inserts each one at a fixed position: on the line above the statement for shell and Python execution, whose code argument is usually long or spans several lines, and at the end of the line that ends the call for all other primitives. A call in the condition of an `if` or `while` statement gets its comment at the end of the header line, because the runtime's lookup treats the whole compound statement as the call's statement and would otherwise find a comment from the body first. We accept the annotated plan only if its abstract syntax tree is unchanged and the runtime's lookup returns the new comment for every annotated call; otherwise we call the model again, up to four attempts in total, and keep the original plan if none passes. We ran this procedure on the final code policies of OSWorld with the Terra converter (348 plans) and of ScienceBoard with the Opus 5 and Terra converters (159 and 167 plans), and it annotated 673 of the 674 plans with 4,321 comments; the remaining plan does not parse as Python.

Verifier inputs. All runs send a fixed note stating that the machine runs Ubuntu with GNOME, where Ctrl- and Alt-based shortcuts are native, because the task instructions given to the verifier do not name the operating system and the rubric blocks shortcuts meant for the wrong operating system. We also evaluate two optional additions to the production rubric. The *exemptions* address correct plans, whose close, kill, save, and retry steps the production rubric reads as destructive or already done. They state that the following are not by themselves reasons to block: opening or focusing an application, window, or file that is already open; closing or killing an application when the explanation gives a task-related reason, unless the screen shows unsaved work that the step would discard; saving the open document; and repeating a step that the explanation says is a retry, or re-navigating to the page already shown. The *blocked targets* add two reasons to block that the production rubric missed on perturbed initial states: a shortcut or typed text meant for a specific application while that application is closed, minimized, not in front, or has no document loaded; and any input aimed at a window that a dialog, popup, or menu covers, unless the input is aimed at the dialog itself. With blocked targets, the rubric's allowance for targets not visible in the screenshot no longer covers these two cases. We report results with and without these additions, since both change the rubric used in production.

B.5 OSWorld-V2 extension

This section records the details of the OSWorld-V2 experiment (Section 4.5) that differ from the OSWorld-Verified and ScienceBoard setup, together with the infrastructure events that affected the runs.

Benchmark version and task set. We use the June 2026 release of OSWorld-V2 (`v2026.06.24`), with task definitions, evaluators, and assets unmodified. We report 107 of the 108 tasks. We exclude task 041 because its instruction hard-codes the benchmark authors' GitLab instance, while setup and grading use the locally deployed one, so no run can be scored as intended. Task 093 is included as a zero for our method, because the converter did not produce a program for it.

Website hosting. 39 of the tasks use web applications that were originally hosted by the benchmark authors. During our experiments, the authors stopped serving the `v2026.06.24` version of these applications and replaced it with a newer version at the same address. The switch happened between 2026-08-25 and 2026-09-10.

Base agent. The base agent is a custom harness driving Claude Opus 5 with maximum adaptive thinking and a budget of 500 steps per task, the budget that the benchmark protocol specifies for Claude models. Its seed trajectory for each task comes from run 1.

Policy construction. The loop follows the refinement algorithm with the settings in Table 10. It differs from the OSWorld and ScienceBoard runs in four ways. First, the converter is an agent (Claude Opus 5) that rewrites the seed trajectory into a program, replacing pixel coordinates with element descriptions and deriving task values at run time instead of copying them from the trajectory. Second, the continuation agent is the custom harness running DeepSeek-V4.1-Flash, with 500 steps and a three-hour wall-clock cap, instead of the base model. Third, the step-level judge receives each rollout as a single transition: the screenshots before and after execution, and the program's output, capped at its last 50 log entries. It therefore returns a verdict and a failure mode but cannot localize the failure to a step. Fourth, the healer also rewrites the policy after a failure in the last iteration, and the rewritten policy is returned without being judged. As in the other benchmarks, the loop skips the judges and the continuation for rollouts that end in a code execution error.

Table 10: Settings of the OSWorld-V2 construction runs.

Setting	Value
Iterations N / rollouts per iteration K / final trials	5 / 3 / 3
Converter Φ_{conv} and healer Φ_{heal}	Claude Opus 5
Continuation agent μ	harness + DeepSeek-V4.1-Flash, 500 steps, 3 h cap
Task-completion judge J_{task}	Claude Sonnet 4.6, 60 tool calls, 900 s
Observer / localizer	Gemini 2.5 Flash / Gemini 2.5 Pro
Screenshots per heal	5, evenly spaced
Construction budget	\$60 per task (never reached; max \$27.66)
Task wall-clock limit	10 h (exceeding it counts as an infrastructure failure)

Infrastructure reruns. We rerun a task only when an infrastructure fault affected its result, never because of a low score, and every completed rerun replaces the original score, including when the score goes down. Task instructions, rubrics, and score weights are unchanged. We ran 13 reruns on tasks with infra failures (004, 005, 010, 018, 021, 024, 025, 058, 061, 073, 089, 092, 100). Of these, four scores went up (004, 025, 092, 100), one went down (061), and eight were unchanged. Tasks whose reruns are still running keep their original scores.

Cost. Construction cost \$8.2 per task on average in LLM calls (Opus 5 58%, Sonnet 4.6 30%, DeepSeek 9%, Gemini 3%) and a median of 2.9 VM-hours per task. Counting infrastructure reruns, total LLM spend was $1.8\times$ that amount. Grounding calls and LLM calls inside official evaluators are not metered. We estimate script costs during execution due to LLM calls to be \$0.03–\$0.08 on average.

C Constructing Policies from Unsuccessful Trajectories

To examine whether NSPI can build a working policy from an unsuccessful trajectory. We split the runs by whether the selected seed trajectory τ_μ succeeds under the benchmark evaluation script. A failed seed is from tasks on which the base agent fails all three attempts. The resulting split represents how NSPI behaves when it starts from a successful versus an unsuccessful demonstration of the task.

We find that NSPI can construct reliable policies from unsuccessful trajectories to a substantial degree as shown in Figure 4. Across the 356 policies built from a failed seed, 62 succeed in all three final runs under the benchmark evaluator, reflecting policies that are consistently correct rather than occasionally lucky. On tasks with a successful seed, the final plan also improves over the base agent in three of the four settings, indicating that NSPI generally adds reliability even when a correct demonstration is already available.

Recovering from a failed seed is more costly than refining a successful one. Failed seeds cost 18–31% more and take 17–38% longer to reach a policy than successful seeds, and in three of four settings they receive a larger share of judge rejections during policy iteration . This is consistent with the intuition that a failed trajectory contains more incorrect or missing steps that must be diagnosed and repaired before the resulting policy passes the judges.