HARNESSNET: FROM ENGINEERED HARNESSES TO UNIFIED NEURAL FEEDBACK CONTROL

Anonymous authors
Paper under double-blind review

ABSTRACT

An LLM agent is shaped not only by its language model, but also by the *harness* that runs it: the control logic that decides what context the model sees, which proposed commands are executed, when to call the model again, and when to stop. These decisions are central to long-horizon agent performance, yet existing harnesses are typically hand-engineered or obtained through search over harness programs, and remain fixed after deployment. We introduce HarnessNet, a neural feedback controller that learns the harness while keeping the LLM frozen. At each step, HarnessNet reads the interaction history and emits one executable control: request the LLM, execute a previously proposed command, read an environment output, or finish. Controls can reference earlier replies or observations instead of regenerating them, allowing long outputs to be reused exactly. The controller is first trained to imitate an engineered harness, then improved from task success by comparing multiple attempts on the same task, without gradients through the LLM or environment. On AppWorld, HarnessNet matches and then surpasses engineered control. Imitation alone reaches 86.0% success on all 57 development tasks, matching the Meta-Harness baseline under the same budgets. Two reward-driven updates from only 48 training attempts raise success to 91.2%, outperforming both the imitation controller in the same evaluation and the engineered harness. The newly solved tasks mainly come from cases where imitation previously produced invalid controls or exhausted its interaction budget, showing that learning improves how the agent is run rather than the frozen LLM itself.

1 INTRODUCTION

An LLM agent is more than its language model. To complete a long-horizon task across applications, such as settling a shared bill, the model may be able to write the right API calls, but another component must decide what the model sees, which proposed commands are executed, how failures are handled, and when the task should stop. This surrounding control program is the *harness*. It coordinates the model over many interaction steps and can determine task success even when the underlying LLM is unchanged. Recent work has shown that agents can be substantially improved by modifying only this harness, through better reusable context or automated search over harness code (Zhang et al., 2025; Lee et al., 2026).

Existing harnesses, however, are still engineered rather than learned. Their knowledge about how to run an agent is encoded in prompts, rules, and code written by humans or proposed by a search procedure. Once deployed, such a harness remains fixed: it does not improve from the outcomes of the tasks it executes, and further gains require another round of manual design or search. This raises a natural question: can the harness itself be turned into a neural controller that is initialized from an existing harness and then improves from task outcomes, while the LLM remains frozen? This is challenging for three reasons. The LLM and environment are black boxes, so gradients cannot pass through them. Feedback is usually delayed until the end of a long episode that may contain hundreds of control decisions. Moreover, every controller output must be executable, since a single malformed control can terminate the task.

We introduce HarnessNet, a learned neural harness for controlling a frozen LLM agent (Figure 1). At each step, the controller reads the interaction history and emits one executable *control*: request the LLM, execute a command previously proposed by the LLM, read an environment output, or finish

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.

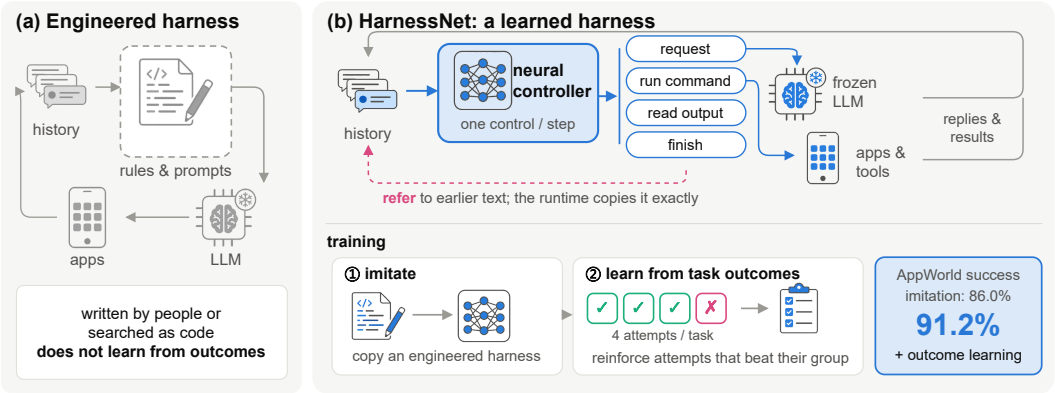

Figure 1: **HarnessNet turns the harness into a learned controller.** (a) An engineered harness: rules and prompts, written by people or found by code search, decide what the LLM sees and which commands run, and they stay fixed. (b) HarnessNet: one neural controller reads the history and issues one control per step to the frozen LLM and the apps; controls can refer to earlier text, which the runtime copies exactly. The controller first imitates an engineered harness and then reinforces the attempts that beat their group, reaching 91.2% success on AppWorld.

the task. Two design choices make this policy practical to learn. First, controls can refer to earlier replies or observations instead of regenerating them. The runtime then copies the referenced content exactly, allowing long model outputs and tool results to be reused without corruption. Second, each control is factorized into an operation, which specifies what to do, and arguments, which specify how to do it. This factorization exposes the agent-level decision itself as a target for learning and exploration.

Training proceeds in two stages. HarnessNet first uses behavioral cloning (BC) to imitate an existing harness, acquiring reliable request construction, command dispatch, observation reuse, and stopping behavior. It is then improved from task outcomes: for each training task, the controller produces several attempts, the task evaluator assigns binary success or failure, and attempts that outperform their group are reinforced (Ahmadian et al., 2024; Shao et al., 2024). Only the controller is updated; no gradient through the LLM or environment is required. For the hidden-state variant, which routes decisions through learned internal control states, a zero-initialized residual ensures that reinforcement learning starts exactly from the cloned policy.

We evaluate HarnessNet on AppWorld (Trivedi et al., 2024). The behavioral-cloning controller completes 49 of 57 development tasks, matching the engineered Meta-Harness at 86.0% success under the same budget. Outcome-based learning then improves beyond imitation: two updates, computed from only 48 training attempts on task families outside the development set, raise success from 82.5% to 91.2% in a matched evaluation, with seven tasks gained and two lost. The learned controller therefore outperforms both its own imitation starting point and the engineered harness. Error analysis shows that six of the seven newly solved tasks are cases where the imitation controller had produced an unusable control or exhausted its interaction budget. Thus, feedback improves how the agent is run rather than the frozen LLM itself. We further show that, after imitation, ordinary sampling rarely changes the selected operation, while a separate operation temperature expands operation-level exploration from zero to 15 of 24 probe states. Our contributions are threefold:

- We formulate the agent harness as a learnable neural feedback policy over a frozen LLM, making the control layer of an agent the object of learning.
- We design HarnessNet, which combines reference-based controls, operation–argument factorization, imitation initialization, and outcome-based reinforcement learning, including an initialization-preserving residual for the hidden-state controller.
- We show on AppWorld that a learned harness can match an engineered one and then improve to 91.2% success from task feedback. We further analyze which control behaviors change and how exploration can be directed at the level of harness decisions.

2 Related Work

Agent interaction and harness optimization. ReAct organizes model reasoning around environment interaction (Yao et al., 2022). More recent methods optimize reusable context or the software that governs this interaction: ACE evolves context from experience, and Meta-Harness searches over harness implementations (Zhang et al., 2025; Lee et al., 2026). These methods show that substantial agent behavior resides outside the LLM weights. HarnessNet studies a complementary learning object: a parameterized policy that selects harness controls from the evolving history. A frozen harness provides an initialization target and a practical execution reference.

Learning policies around a black-box model. Advisor Models learn to steer a frozen worker, and prompting policies distill experience into multi-step guidance (Asawa et al., 2025; Sayana et al., 2026). HarnessBridge learns observation projection and action admission with a shared policy; offline harness RL learns structural execution actions from terminal rewards (Wang et al., 2026; Yi & Song, 2026). EvoHarness-RL studies learned interactions with an external workspace (Ning et al., 2026). HarnessNet places context selection, requests, dispatch, and stopping behind one control interface. Its specific focus is the factorization of history into control states and of those states into an operation and its executable arguments.

Reinforcement learning for LLM agents. Reinforcement learning has become a central tool for improving LLMs, from reasoning with group-relative objectives (Shao et al., 2024; DeepSeek-AI, 2025) to multi-turn agents trained in web, game, and app environments (Zhou et al., 2024; Qi et al., 2025; Wang et al., 2025; Chen et al., 2025); LOOP, for example, trains the acting model directly on AppWorld (Chen et al., 2025). These methods update the weights of the LLM that acts. Closer to our setting, RLPrompt and Retroformer train a smaller model with policy gradients to steer a frozen LLM through prompts or reflections (Deng et al., 2022; Yao et al., 2024). HarnessNet instead learns the complete harness policy, covering context construction, model invocation, command dispatch, and stopping, while the LLM remains a frozen black box.

Continuous conditioning and policy optimization. Prefix-tuning and prompt tuning learn continuous inputs for accessible language models (Li & Liang, 2021; Lester et al., 2021); HarnessNet uses continuous conditioning only inside a local compiler whose output is an executable control, so the remote LLM remains a black box. This distinction is important because the learned continuous state never needs to be interpreted by, or backpropagated through, the deployed LLM. Latent-action policies similarly separate decisions from their language realization (Zhao et al., 2019; Jia et al., 2025), while HarnessNet grounds this separation in executable harness controls. Behavioral initialization connects to imitation learning (Ross et al., 2011), outcome-based updates to policy-gradient methods (Sutton et al., 1999; Schulman et al., 2017), and the overall formulation to stochastic control (Bertsekas, 2019; Kaelbling et al., 1998).

3 Harnesses as Feedback Policies

State, history, and control. Let x be a task, e_t the execution environment, and h_t the causal history available before decision t. The history contains the task description, LLM replies, environment observations, previous controls, and remaining budgets. A harness policy chooses a serialized control $u_t \sim \pi_\Theta(\cdot \mid h_t)$. Each control consists of an operation and its arguments, such as a new model request, a command dispatch, or a stopping decision. The policy therefore acts on the organization of the agent's computation as well as its interaction with the environment.

The materializer Φ resolves references in a control against the available history. The resulting action a_t is processed by the frozen LLM q or execution kernel K:

$$u_t \sim \pi_\Theta(\cdot \mid h_t), \qquad a_t = \Phi(h_t, u_t), \qquad h_{t+1} \sim P_{q,K}(\cdot \mid h_t, a_t). \tag{1}$$

A request operation invokes q and records its response. Read and execution operations interact with K; a finish operation terminates the episode. Responses and execution outcomes become available to the next decision. This gives all operations a common feedback loop while allowing the controller to use an existing model output without requesting it again.

Control responsibility. The boundary between policy and kernel determines what is being learned. The policy selects the information to send, the request content, the operation, and its timing. The

kernel resolves named resources, checks permissions and budgets, and executes the declared action. The kernel runs only commands that the LLM produced earlier in the episode. A request may reuse text generated by the controller earlier in the same trajectory. Consequently, reusable strategy can reside in controller parameters and causal history, rather than in a hand-written playbook.

Trajectory objective. We evaluate decisions by their consequences for the entire task:

$$\mathcal{J}(\Theta) = \mathbb{E}_{\tau \sim \pi_\Theta, P_{q,K}} \left[R(\tau) - \sum_{t<T} c_t \right]. \tag{2}$$

Here R measures the task outcome and c_t represents any stage costs included in the learning objective. The experiments use outcome rewards under fixed execution budgets. Those budgets also enter h_t, allowing the controller to condition its decisions on the remaining opportunities to interact. The same objective connects information acquisition, execution, and stopping: the value of a control includes the future decisions it enables.

An observation that changes a later request can therefore have value even if it does not immediately change task state, while an unnecessary call consumes an opportunity to act later. HarnessNet learns these control decisions from history rather than requiring a hand-designed Markov state.

4 HARNESSNET

HarnessNet has three parts: an encoder that summarizes the interaction history into a control state, a compiler that turns this state into an executable control, and a two-stage training procedure of imitation followed by reinforcement learning (Algorithm 1).

4.1 FROM INTERACTION HISTORY TO CONTROL STATES

HarnessNet places the decisions of the harness behind a shared representation. Given history h_t and control-register inputs R, an encoder constructs

$$H_t = F_\theta(h_t; R) \in \mathbb{R}^{m \times d}, \qquad Z_t = A_\phi(H_t). \tag{3}$$

The evaluated model uses $m = 128$ control positions. The encoder integrates observations and previous decisions; the adapter maps their hidden states to the conditioning space of a local compiler. The compiler receives a static description of the operation interface and the state-dependent prefix Z_t. This arrangement gives different harness operations a common information source and a common pathway for learning.

The registers represent the current decision, while the compiler emits an operation with arguments of variable length using its inherited language and protocol knowledge.

HarnessNet admits two instantiations of this policy. A *direct-text* controller generates the serialized control directly from the history with a LoRA-adapted language model (Hu et al., 2021); the *hidden-state* controller routes the decision through H_t and the compiler below. Both operate the same request, reference, and execution primitives and are trained with the same imitation and outcome objectives. The direct-text controller is used for task-scale experiments; the hidden-state controller makes the intermediate control representation and its update explicit.

4.2 COMPILING DECISIONS INTO EXECUTABLE CONTROLS

The compiler factors a control into operation selection and argument generation:

$$\pi_\Theta(u_t \mid h_t) = g_\omega(o_t \mid H_t) \prod_{j=1}^{L_t} D_\psi(b_{t,j} \mid Z_t, o_t, b_{t,<j}), \qquad u_t = (o_t, b_t). \tag{4}$$

The operation factor selects what to do. The autoregressive factor supplies its content, such as a request body or a reference to an existing command. The operation head uses pooled register features; the argument decoder uses a soft prefix and hidden-state modulation.

References separate selection from transport. A control can contain literal text, a reference to an earlier observation, or a composition of available resources. The materializer expands the selected

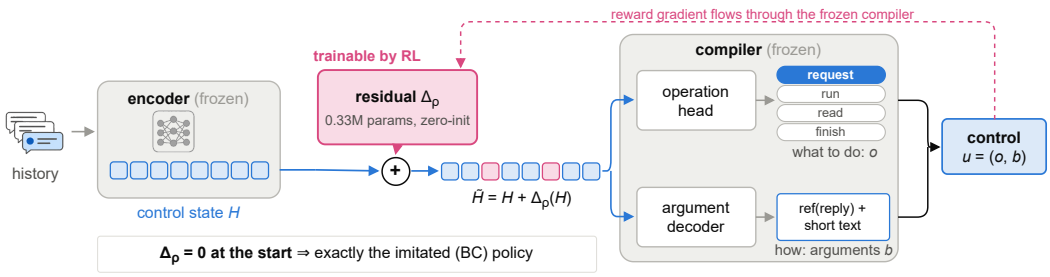

Figure 2: **Reward learning changes only a small residual on the control state.** The encoder and compiler are frozen after imitation. A trainable residual Δ_ρ perturbs the control state H, and the perturbed state $\widetilde{H}$ feeds both the operation head (what to do) and the argument decoder (how to do it). Reward gradients reach the residual through the frozen compiler. Because $\Delta_\rho = 0$ at the start, learning begins exactly from the BC policy.

resources after generation. For example, a controller can construct a new request from its current instruction and a prior execution result. Its learned decision is which result to include and how to use it; the recorded result is copied exactly by the runtime. This division keeps information selection inside the policy while reusing the environment's existing representation of the data.

4.3 Behavioral Initialization

An existing harness supplies demonstrations (h_t, u_t^*), fitted by maximum likelihood:

$$\mathcal{L}_{\mathrm{BC}}(\Theta) = -\mathbb{E}_{(h,u^*)\sim\mathcal{D}} \log \pi_\Theta(u^* \mid h). \tag{5}$$

The targets capture how the harness operates the LLM: its request assembly, resource selection, and execution decisions. Corrective demonstrations on histories generated by the controller itself follow dataset aggregation (Ross et al., 2011).

This initialization transfers a useful coordination policy before outcome-based optimization begins. It is particularly important for long-horizon tasks: early interface errors can prevent a controller from reaching the states at which higher-level choices matter. We retain a frozen BC checkpoint as the reference policy throughout evaluation. The initialization lineage and fitting configuration are given in Appendix B.

4.4 An Initialization-Preserving Control Residual

For hidden-state reward learning, we freeze the BC encoder, adapter, operation head, and compiler. A state-conditioned residual modifies the control states:

$$\begin{aligned} \widetilde{H}_t &= H_t + \Delta_\rho(H_t), \\ \Delta_\rho(H) &= \alpha s(H) \odot \tanh\Big(W_{\mathrm{up}}\,\mathrm{GELU}(W_{\mathrm{down}}\,\mathrm{LN}(\widehat{H}))\Big). \end{aligned} \tag{6}$$

Biases are implicit. $\widehat{H}$ uses fixed BC feature standardization, and $s(H)$ is a detached per-register RMS scale. The residual has rank 64, amplitude $\alpha = 0.05$, and 330,304 trainable parameters. Its output feeds both the operation head and the argument compiler (Figure 2). Thus one learned perturbation can coordinate what operation is chosen and how that operation is expressed.

The final projection and bias initialize to zero. For fixed sampling and execution rules, this implies:

$$\Delta_{\rho_0}(H) = 0 \implies \pi_{\rho_0}(u \mid h) = \pi_{\mathrm{BC}}(u \mid h) \implies p_{\rho_0}(\tau) = p_{\mathrm{BC}}(\tau). \tag{7}$$

The first implication follows because both compiler factors receive the same inputs. The second follows by multiplying the equal policy factors with the shared conditional transition kernels. This gives reward learning a precise starting policy, including its interaction with a stochastic LLM.

The elementwise bound $|\Delta_\rho(H)_{ij}| \le \alpha s(H)_i$ sets the scale of the representation change relative to each BC register. Freezing the compiler weights preserves its learned realization mechanism while

allowing gradients through its computation into the residual. The reward-trainable module is thus small relative to the base model, and its changes stay attached to a well-defined control state.

Credit through the compiler. Let s_o and s_b denote the log probabilities of the sampled operation and arguments, respectively. With J_ρ the Jacobian of the perturbed control states with respect to the residual parameters, the local score decomposes as

$$\nabla_\rho \log \pi_\rho(u \mid h) = J_\rho^\top \left(\nabla_{\widetilde{H}} s_o + \nabla_{\widetilde{H}} s_b \right). \tag{8}$$

Both signals act on the same residual, even though their compiler weights are fixed. This lets us measure whether task feedback reaches operation selection and tune its exploration separately.

4.5 Outcome-Based Optimization

For remaining return G_t and a history-dependent baseline $b(h_t)$, the score-function gradient is

$$\nabla_\Theta \mathcal{J} = \mathbb{E}\left[\sum_{t<T} \nabla_\Theta \log \pi_\Theta(u_t \mid h_t)(G_t - b(h_t)) \right]. \tag{9}$$

Only the controller likelihood is differentiated. The frozen model and execution environment determine the sampled returns, so their gradients are unnecessary. The probability belongs to the control that was generated, including its operation and arguments; expansion of a reference is part of the transition. Appendix A gives the trajectory derivation.

Group-relative advantages. For each training task, we sample a group of $G = 4$ trajectories from the current controller and score them with the binary evaluator outcome R_i. We consider two group baselines: the leave-one-out advantage $A_i = R_i - \frac{1}{G-1} \sum_{k \neq i} R_k$ (RLOO; Ahmadian et al., 2024) and the group-normalized advantage $A_i = (R_i - \bar{R})/(\sigma_R + \epsilon)$ (GRPO; Shao et al., 2024). Both require no learned critic, which is attractive when a single long-horizon episode contains hundreds of controls. Groups whose members share the same outcome carry no relative signal and are skipped. Each advantage multiplies all controller tokens of its trajectory, averaged over the trajectory's tokens, inside a clipped PPO surrogate that uses the recorded behavior probabilities (Schulman et al., 2017).

Conservative, verified updates. One parameter update changes every decision of a long episode, so we keep learning close to the inherited competence. A small imitation term (weight 0.05) on training-task demonstrations keeps the controller close to its initialization, analogous to the pretraining mixture used in RLHF (Ouyang et al., 2022). A candidate update is accepted only if its sampled KL and clipping diagnostics stay within fixed trust bounds and the surrogate improves; otherwise parameters and optimizer state are restored and training stops. The result is a small number of conservative, verified policy updates. For the hidden-state residual, whose initialization already preserves the BC policy exactly, we use no imitation term and a fixed token divisor (4096), which keeps shared early decisions from receiving length-dependent weights (Appendix A).

Exploration at the compiler interface. Equation 4 also separates two sources of sampling variation. The operation distribution can use temperature β_o, while argument generation uses β_b. Raising β_o changes the range of operations considered without simultaneously heating every token of their arguments. The factorization therefore provides both a learning interface and a way to localize exploration, which we measure in Section 5.

5 Experiments

We evaluate whether the harness can be learned as a control policy around a frozen LLM. We ask three questions: whether behavioral cloning can inherit a competitive engineered harness; whether task-level outcome feedback can improve the controller beyond imitation; and what learning changes in the controller's operation choices, execution behavior, and interaction budget use.

5.1 Experimental Setup

We use AppWorld (Trivedi et al., 2024), where agents complete everyday tasks by operating applications through executable interactions. The development set contains 57 tasks from 19 task

Algorithm 1 HarnessNet initialization and outcome-based learning

Require: Frozen LLM q, kernel K, training tasks, harness demonstrations
1: Fit the controller π_Θ by behavioral cloning (Eq. 5)
2: Hidden-state variant: freeze BC components; add Δ_ρ with zero output projection
3: **for** each update round **do**
4: **for** each training task **do**
5: Sample G trajectories with π_Θ; record behavior probabilities
6: Score outcomes with the evaluator; compute group-relative advantages
7: **end for**
8: Propose a clipped policy-gradient update (plus an imitation term for direct-text)
9: Accept if trust checks pass and the surrogate improves; otherwise restore and stop
10: **end for**
11: Evaluate the final controller from task start on development tasks

Table 1: Task success on the complete AppWorld development set (57 tasks). Within each evaluation round, all controllers share tasks, seeds, the frozen LLM, and execution budgets. Wins/losses are paired against the BC controller of the same round.

Round	Controller	Success	Rate (%)	Wins/losses vs. BC
I	Meta-Harness (engineered)	49/57	86.0	3/3
	HarnessNet (BC)	49/57	86.0	—
II	HarnessNet (BC)	47/57	82.5	—
	HarnessNet + RLOO (6 updates)	49/57	86.0	8/6
	HarnessNet + GRPO (2 updates)	**52/57**	**91.2**	**7/2**

families with three variants each. Controllers derive from Qwen3.5-4B with LoRA adapters (rank 64) and operate a frozen `deepseek-v4-flash` LLM with temperature zero and an 8,000-token response limit. Each task allows 40 LLM requests, 80 environment executions, 256 controls, and 1,800 seconds. The metric is evaluator task success. The engineered reference is an adapted frozen Meta-Harness artifact (Lee et al., 2026), run under the same budgets.

The task-level experiments use the direct-text controller. Reinforcement learning starts from its BC checkpoint and uses 12 training tasks drawn from families disjoint from the development set. Each task contributes four rollouts at sampling temperature 1.0, giving 48 training trajectories in total. RLOO and GRPO training share these rollouts, the imitation data, the minibatch order, and all trust bounds, and differ only in their advantage estimator. No development task is used for gradients. All controllers are evaluated from the initial task state at temperature 0.7; within an evaluation round, all controllers share tasks, seeds, and budgets. Further details are given in Appendix B.

5.2 Main Results

A neural controller inherits a competitive harness. In Round I of Table 1, the BC controller completes 49 of 57 tasks, exactly matching the engineered Meta-Harness. The two agents solve different subsets: each succeeds on three tasks the other misses. The student thus acquires a competitive coordination policy without reproducing every Meta-Harness trajectory, and it does so with similar reliance on the LLM (1,249 versus 1,260 LLM requests over the development set). Request construction, history reuse, dispatch, and stopping are all carried out by one learned policy through the shared control interface.

Task feedback improves the controller beyond imitation. Round II compares learned controllers with their own initialization under identical conditions. Figure 3a shows that GRPO training raises success from 82.5% to 91.2%, a gain of 8.8 points, with seven tasks gained and only two lost. The resulting controller also exceeds the 86.0% of the engineered harness. This improvement comes from two accepted updates computed on 48 training rollouts: a small amount of outcome feedback, from training tasks outside the evaluated families, suffices to move a harness policy that was fitted to imitate an engineered one. RLOO, trained on the same rollouts, reaches 86.0%. With identical data, group normalization yields the larger gain with fewer accepted updates. Normalization rescales

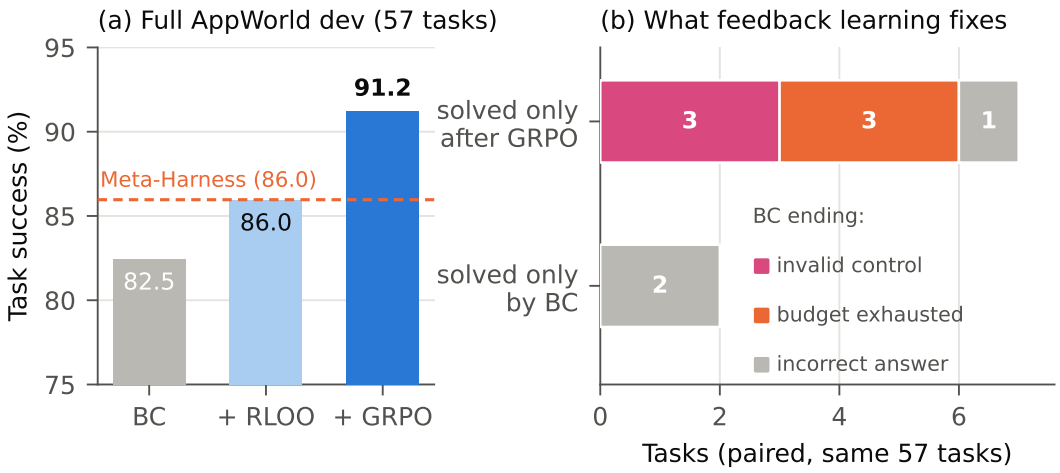

Figure 3: **Task feedback lifts the neural harness beyond imitation and beyond the engineered harness.** (a) Success on all 57 AppWorld development tasks under identical tasks, seeds, and budgets. Two GRPO updates raise the BC controller from 82.5% to 91.2%; the dashed line marks the engineered Meta-Harness. (b) Paired outcomes against BC. Six of the seven tasks newly solved after learning are ones on which BC ended with an invalid control or an exhausted interaction budget.

each group by its outcome spread, which enlarges the update from groups in which a single rollout contrasts with the other three.

5.3 What Does Feedback Learning Change?

Figure 3b decomposes the paired outcomes. Of the seven tasks that only the learned controller solves, BC had terminated three with an invalid control and three by exhausting its LLM-request or time budget; only one was an incorrect answer returned normally. Across the development set, episodes that the controller closes deliberately rise from 51 to 55, while terminations by invalid control or exhausted budget fall from six to two. Feedback learning thus improves how the controller operates the agent: it issues controls the runtime can execute, and it reaches a decision within the available interactions. Both properties are harness competences rather than capabilities of the frozen LLM, which is unchanged across all controllers.

The gains are also coherent at the family level. All three variants of one task family, each failed by BC, are solved after learning; the variants differ in content but share a coordination structure. Learning repairs all three, indicating that it changed how the controller handles this shared structure. On one newly solved task, the learned controller succeeds with 18 LLM requests where BC spent 36 without success, illustrating that better coordination can also make the use of the LLM more direct.

5.4 Directing Exploration at the Level of Operations

Group-relative learning can only reinforce behavior that sampling proposes. The factorized compiler of the hidden-state controller lets us measure this at the level of operations. In a sampled hidden-state training batch, only two of 1,187 controls select an operation different from the BC greedy choice, and at 20 designated high-temperature positions the total probability of an alternative operation is 6.66×10^{-6}. After imitation, token-level sampling varies how a control is written far more than which operation is chosen.

The operation factor in Equation 4 provides a direct remedy. We calibrate the factorized sampler on 24 training states with eight samples each, holding argument temperature at 0.3. Figure 4 shows that at an operation temperature of 0.3, none of the states has two distinct legal operations; raising it to 3 and 5 expands this coverage to nine and 15 states. The legal-control rate stays essentially unchanged (120/192 versus 122/192). Operation diversity and argument randomness can therefore be set independently, which places exploration at the level where the policy decides how to use the agent: whether to request, inspect, dispatch, or finish.

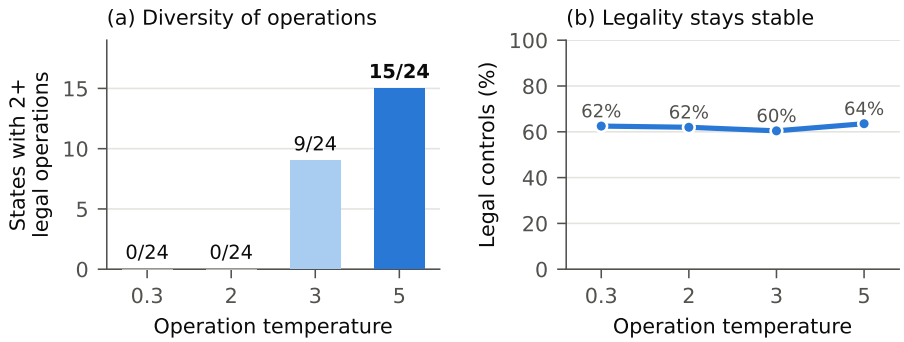

Figure 4: **Operation temperature increases decision-level exploration without destabilizing control generation.** We calibrate the factorized sampler on 24 training states with eight samples per state, while keeping the argument temperature fixed at 0.3. (a) Increasing the operation temperature expands the number of states in which sampling produces two or more distinct legal operations, from 0/24 at low temperature to 15/24 at temperature 5. (b) The legal-control rate remains stable across the same range, indicating that operation-level exploration can be increased without substantially degrading control validity.

6 Discussion

Why limited feedback can suffice. Imitation gives the controller the basic competence needed to run an episode: executable controls, valid references, and a reasonable order of operations. Outcome feedback then only needs to adjust the decisions that separate success from failure. By comparing multiple attempts on the same task, the learning signal focuses on choices such as recovering after a rejected command or stopping before the budget is exhausted. This helps explain why only 48 rollouts and a few conservative updates can change task outcomes.

Engineered harnesses as teachers. HarnessNet does not replace harness engineering or search; it uses them as teachers. Hand-written or searched harnesses provide reliable demonstrations, while outcome feedback adapts their behavior to cases the fixed program did not handle well. In this sense, engineering, search, and learning are complementary: engineering supplies the starting policy, and learning improves it from execution outcomes.

Limitations and outlook. Experiments focus on AppWorld with one frozen LLM, enabling a controlled comparison under matched budgets. Broader evaluation across environments, tools, and model families is needed to test transfer. Since HarnessNet interacts with the LLM only through requests and responses, the formulation is not tied to a specific model implementation. Larger task sets, longer training, and more targeted operation-level exploration are natural directions for extending the observed gains further.

7 Conclusion

This work considers whether the control layer of an LLM agent can be learned directly from experience. To this end, we introduce HarnessNet, a neural harness that turns hand-written control logic into a learned feedback controller while keeping the underlying LLM frozen. The controller reads the interaction history, issues executable controls that can refer to earlier text, and separates what to do from how to do it. It is first initialized by imitating an engineered harness and then improved from task outcomes, without gradients through the LLM or environment. On AppWorld, imitation alone matches the engineered Meta-Harness at 86.0% success, and two reward-driven updates from 48 training rollouts raise success to 91.2%, mainly by reducing invalid controls and budget exhaustion. Together, these results show that the harness is not merely implementation detail: learning how to request the model, execute proposed commands, reuse observations, and stop can improve long-horizon agent performance without updating the LLM. This suggests a complementary path for building stronger agents, where progress comes not only from stronger models, but also from learned mechanisms for running them effectively.

AI USE STATEMENT

Generative AI tools assisted code implementation, experimental analysis, and manuscript polishing. The icons in Figures 1 and 2 were generated with an image-generation model. Quantitative plots are generated from recorded measurements. The authors are responsible for the content and interpretation of the work.

ETHICS STATEMENT

Experiments use AppWorld benchmark application data in isolated execution environments. Permissions, execution limits, and checks that executed commands originate from the LLM remain outside the learned controller. Released artifacts should exclude credentials and identifying benchmark metadata.

REPRODUCIBILITY STATEMENT

The appendices specify controller configurations, initialization, task sets, learning objectives, and evaluation procedures. Figures use the same numerical records as the corresponding results. Experiments record checkpoint and source identities; reproduction additionally requires access to the stated capability model and execution environment.

REFERENCES

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting REINFORCE-style optimization for learning from human feedback in LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12248–12267, 2024.

Parth Asawa, Alan Zhu, Abigail O'Neill, Matei Zaharia, Alexandros G. Dimakis, and Joseph E. Gonzalez. How to Train Your Advisor: Steering Black-Box LLMs with Advisor Models. arXiv preprint arXiv:2510.02453, 2025. URL https://arxiv.org/abs/2510.02453v3.

Dimitri P. Bertsekas. *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019. URL https://web.mit.edu/dimitrib/www/RLbook.html.

Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krähenbühl. Reinforcement learning for long-horizon interactive LLM agents. arXiv:2502.01600, 2025.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric P. Xing, and Zhiting Hu. RLPrompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. arXiv preprint arXiv:2106.09685, 2021. URL https://arxiv.org/abs/2106.09685v2.

Chengxing Jia, Ziniu Li, Pengyuan Wang, Yi-Chen Li, Zhenyu Hou, Yuxiao Dong, and Yang Yu. Controlling large language model with latent actions. arXiv preprint arXiv:2503.21383, 2025. URL https://arxiv.org/abs/2503.21383.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998. doi: 10.1016/S0004-3702(98)00023-X.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-End Optimization of Model Harnesses. arXiv preprint arXiv:2603.28052, 2026. URL https://arxiv.org/abs/2603.28052v1.

Brian Lester, Rami Al-Rfou, and Noah Constant. The Power of Scale for Parameter-Efficient Prompt Tuning. arXiv preprint arXiv:2104.08691, 2021. URL https://arxiv.org/abs/2104.08691v2.

Xiang Lisa Li and Percy Liang. Prefix-Tuning: Optimizing Continuous Prompts for Generation. arXiv preprint arXiv:2101.00190, 2021. URL https://arxiv.org/abs/2101.00190v1.

Xuying Ning, Dongqi Fu, Tianxin Wei, Hanqing Zeng, Yuanchen Bei, Bingxuan Li, Zihao Li, Qifan Wang, Xiang Shen, Yifan Wu, Jiayi Liu, Hong Li, Yinglong Xia, Xiangjun Fan, Hanghang Tong, and Jingrui He. EvoHarness-RL: Learning Self-Evolving Runtime Harness for Long-Horizon LLM Agents. arXiv preprint arXiv:2608.05446, 2026. URL https://arxiv.org/abs/2608.05446v1.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, Tianjie Zhang, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations*, 2025.

Stephane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pp. 627–635, 2011. URL https://proceedings.mlr.press/v15/ross11a.html.

Krishna Sayana, Ketan Todi, and Ambarish Jash. Prompting Policies for Multi-step Reasoning and Tool-Use in Black-box LLMs with Iterative Distillation of Experience. arXiv preprint arXiv:2605.14443, 2026. URL https://arxiv.org/abs/2605.14443v1.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347, 2017. URL https://arxiv.org/abs/1707.06347v2.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. arXiv:2402.03300, 2024.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2 edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12, 1999. URL https://papers.nips.cc/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. URL https://arxiv.org/abs/2407.18901.

Xiaoxuan Wang, Haixin Wang, Alexander Taylor, Jason Cong, Yizhou Sun, and Wei Wang. HarnessBridge: Learnable Bidirectional Controller for LLM Agent Harness. arXiv preprint arXiv:2606.12882, 2026. URL https://arxiv.org/abs/2606.12882v1.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, et al. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. arXiv:2504.20073, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv preprint arXiv:2210.03629, 2022. URL https://arxiv.org/abs/2210.03629v3.

Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh Murthy, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, Ran Xu, Phil Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. Retroformer: Retrospective large language agents with policy gradient optimization. In *International Conference on Learning Representations*, 2024.

Haiwen Yi and Xinyuan Song. Learning to Control LLM Agent Harnesses with Offline Reinforcement Learning. arXiv preprint arXiv:2607.05458, 2026. URL https://arxiv.org/abs/2607.05458v1.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. arXiv preprint arXiv:2510.04618, 2025. URL https://arxiv.org/abs/2510.04618v3.

Tiancheng Zhao, Kaige Xie, and Maxine Eskenazi. Rethinking action spaces for reinforcement learning in end-to-end dialog agents with latent variable models. In *Proceedings of NAACL-HLT*, pp. 1208–1218. Association for Computational Linguistics, 2019. doi: 10.18653/v1/N19-1123. URL https://aclanthology.org/N19-1123/.

Yifei Zhou, Andrea Zanette, Jiayi Pan, Sergey Levine, and Aviral Kumar. ArCHer: Training language model agents via hierarchical multi-turn RL. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62178–62209, 2024.

A Policy Semantics and Gradient Derivation

We derive the outer-policy gradient using standard likelihood-ratio reasoning (Sutton et al., 1999; Sutton & Barto, 2018). Assume a parameter-independent task distribution and fixed conditional LLM, materialization, and environment kernels; differentiable policy probabilities on the sampling support; a bounded horizon; integrable returns; and a valid interchange of derivative and expectation. The transition kernel includes the runtime response to invalid controls.

Writing the trajectory in terms of legal histories and emitted controls gives

$$p_\Theta(\tau) = p_0(h_0) \prod_{t<T} \pi_\Theta(u_t \mid h_t) P(h_{t+1} \mid h_t, u_t). \tag{10}$$

Here P includes mechanical expansion, request or execution dispatch, rejection, and stopping. It is fixed conditional on its arguments, including when a step does not invoke the LLM. For total reward $\mathcal{R}(\tau)$,

$$\nabla_\Theta \mathcal{J} = \int \mathcal{R}(\tau) \nabla_\Theta p_\Theta(\tau)\, d\tau \tag{11}$$

$$= \mathbb{E}\left[\mathcal{R}(\tau) \sum_{t<T} \nabla_\Theta \log \pi_\Theta(u_t \mid h_t) \right]. \tag{12}$$

Past rewards have zero expected contribution to a later action score, and an action-independent baseline can be subtracted because $\mathbb{E}_{u\sim\pi_\Theta}[\nabla_\Theta \log \pi_\Theta(u \mid h)] = 0$. This yields Equation 9. A learned baseline, if used, is a stopped-gradient control variate in the actor objective. Terminal reward is counted once, and padding after absorption has no further policy decision.

Emitted versus expanded controls. If several controls materialize to the same action a, its probability is $\sum_{u:\Phi(h,u)=a} \pi_\Theta(u \mid h)$. The token probability of the expanded request is generally different. We score the sampled control before materialization, yielding a valid estimator for the composed trajectory objective.

The implemented sampling distribution. For argument token logits ℓ, grammar mask χ, and temperature β,

$$\widetilde{p}(k \mid h, w) = \frac{\chi(k; h, w) \exp(\ell_k/\beta)}{\sum_j \chi(j; h, w) \exp(\ell_j/\beta)}. \tag{13}$$

The operation factor uses its own recorded categorical probability. Tokens fixed by the control format have probability one. The learner recomputes these probabilities with the same mask, temperature, and stopping rule used during collection. PPO uses the recorded behavior probabilities to construct its clipped surrogate objective.

Deterministic control registers. For $H = F_\theta(h; R)$ and $Z = A_\phi(H)$, differentiating through a frozen local compiler still yields gradients with respect to Z and H. In the residual experiments only the residual parameters are updated; the cached encoder, registers, adapter, operation head, and compiler weights stay fixed. If instead one samples a latent token sequence v with a parameter-dependent distribution, the joint score must also contain $\nabla \log p_\theta(v \mid h)$. The current deterministic-register residual does not introduce that additional sampling term: its score is fully determined by the compiler operation and argument probabilities.

Group advantage and normalization. For four returns R_i, the leave-one-out advantage is $A_i = R_i - \frac{1}{3}\sum_{k\neq i} R_k$. The fixed-divisor residual objective is

$$\widehat{L} = \frac{1}{4C} \sum_{i=1}^{4} \sum_{j=1}^{L_i} \min(\rho_{ij} A_i, \mathrm{clip}(\rho_{ij}, 1-\epsilon_-, 1+\epsilon_+) A_i), \quad C = 4096. \tag{14}$$

Trajectory-token normalization instead divides each trajectory's token sum by its own L_i. Although $\sum_i A_i = 0$, in general $\sum_i A_i/L_i \neq 0$: identical early controls may receive a nonzero aggregate coefficient because later trajectory lengths differ. A shared divisor keeps the per-token scale constant across group members. At the behavior policy, identical score terms therefore cancel according to $\sum_i A_i = 0$.

B Implementation and Initialization

Table 2: Reinforcement learning of the direct-text controller.

Component	Configuration
Controller	Qwen3.5-4B with LoRA (rank 64, $\alpha = 128$), initialized from BC
LLM	Frozen `deepseek-v4-flash`, temperature 0
Training data	12 training tasks $\times$ 4 rollouts, sampling temperature 1.0
Reward	Binary evaluator success
Advantage	RLOO or GRPO group normalization, $G = 4$
Surrogate	Token-level clipped PPO, $\epsilon = 0.2$, behavior-probability replay
Imitation term	Weight 0.05 on training-task demonstrations
Optimizer	AdamW, learning rate 10^{-6}, gradient clipping 1.0
Updates	Batches of 8 trajectories, at most 6 accepted updates
Trust checks	Sampled KL ≤ 0.01 (mean) and ≤ 0.1 (per action); PPO clip fraction ≤ 0.1
Evaluation	From task start, temperature 0.7, same budgets for all controllers

Table 3: Hidden-state residual configuration. Reward learning updates the residual while preserving the BC encoder and compiler.

Component	Configuration
Encoder family	Qwen3.5-4B initialization with learned control registers
Control positions	128
Compiler input	Control states and a fixed description of the control format
Compiler factors	Learned operation head; autoregressive argument decoder
Conditioning	Soft prefix and final hidden-state modulation
Residual	Rank 64, amplitude 0.05, zero-initialized final projection
Reward-trainable parameters	330,304 residual parameters
Frozen during residual RL	Encoder, registers, adapter, operation head, compiler
Control generation cap	8,192 tokens
Reward	$0.8\,\text{success} + 0.2\,\text{test fraction} - 0.02\min(n_{\text{reject}}, 5)$
Auxiliary losses	None
Residual normalization	Fixed token divisor 4096; complete four-member groups

The operation head standardizes a concatenation of the first register, mean register, last register, and mean of the first eight registers, and predicts the operation; the remaining tokens of the control come from the argument decoder. A reference addresses an earlier resource either by absolute position or relative to the current step, for example the most recent LLM reply. The runtime resolves it to an absolute position before execution and rejects references to missing resources.

The residual standardizes BC features and applies a rank-64 nonlinear update scaled by the RMS of each register. The compiler runs in FP32 to retain numerical precision for small residual updates. BC training adapts decoder LoRA weights (Hu et al., 2021), which stay frozen during residual learning.

Runtime operations. The runtime exposes operations to request an LLM reply, execute a shell command, read or capture terminal output, check whether a process is still running, and finish. Only requests invoke the LLM. The controller decides what each request contains, which command to execute, and when to stop. A command proposed in an earlier LLM reply can be executed without a new request; the runtime executes only commands that the LLM produced earlier in the episode, together with one fixed read-only inspection at the start of each episode.

Initialization. The direct-text controller is initialized by behavioral cloning on demonstrations produced by engineered harnesses (ACE and Meta-Harness; Zhang et al., 2025; Lee et al., 2026), followed by corrective demonstrations on histories generated by the controller itself (Ross et al., 2011). The hidden-state controller is trained on demonstrations from the same source. Reinforcement learning introduces no new demonstrations. Training and evaluation task families are specified in Appendix C.

C Experimental Details and Additional Results

Development set. The 57 development tasks comprise 19 families with three variants each. All episodes are scored by the AppWorld evaluator (`0.1.4.dev0`). Evaluation round I compares the BC controller with Meta-Harness; round II compares the BC controller with the RLOO and GRPO controllers derived from it, using a fresh controller sampling seed. Infrastructure failures (transport errors before a response) are re-run under a fixed replacement policy; policy failures, invalid controls, and exhausted budgets are never re-run.

Reinforcement learning. Training uses 12 tasks from families disjoint from the development set, with four rollouts per task at temperature 1.0 (48 trajectories; 39 successful). Four task groups contain both successes and failures and thus carry group-relative signal. The collection contains 3,380 controller generations and 358,520 sampled tokens, whose behavior probabilities are replayed exactly by the learner before optimization. RLOO accepts six updates; GRPO accepts two, after which its third candidate fails the acceptance test and training stops. The reported GRPO controller is therefore the final checkpoint of its run rather than a checkpoint selected on development outcomes.

Paired outcomes. Against the round-II BC controller, GRPO training wins seven tasks and loses two (48 ties); RLOO wins eight and loses six (43 ties). Wilson 95% intervals for success are $[70.6, 90.2]$ for BC, $[74.7, 92.7]$ for RLOO, and $[81.1, 96.2]$ for GRPO. A paired bootstrap over task families gives a GRPO-minus-BC interval of $[0.0, 22.8]$ points. Episodes closed by the controller rise from 51 (BC) to 55 (GRPO); terminations by invalid control fall from three to one, and budget terminations from three to one.

Operation-level exploration. The calibration of Figure 4 uses 24 predeclared training states (eight request states, eight states just before stopping, and eight states following a runtime rejection), eight samples per state and profile. At operation temperatures $\{0.3, 2, 3, 5\}$ with argument temperature 0.3, legal controls number $\{120, 119, 116, 122\}/192$ and the states with two legal operations $\{0, 0, 9, 15\}/24$. The operation statistics in Section 5 are computed on the 1,187 controls of a sampled hidden-state training batch, using the recorded operation probabilities.

Resource accounting. In round I, BC and Meta-Harness episodes use 1,249 and 1,260 LLM requests. In round II, BC, RLOO, and GRPO use 1,210, 1,325, and 1,288 LLM requests, all within the same per-task budget.

D Evaluation Protocol

Task pairing. Controllers are evaluated on common task conditions, LLM services, and execution budgets. Evaluation begins from the task initial state with the selected checkpoint. Task success is obtained from the environment evaluator. Controller termination and rejection counts are retained as separate measurements.

Outcome inclusion. Invalid policy controls and valid but unsuccessful task outcomes remain in the evaluation as failures. Infrastructure recovery is determined from the failure class and uses a bounded replacement policy; recorded costs include replaced attempts.

Probability consistency. Sampling-probability checks use the actual temperature, operation factor, grammar mask, and stopping convention, so that the probabilities used by the learner match those under which trajectories were collected. Checkpoint and source identities are recorded with the resulting trajectories.

Data scope. The 57-task AppWorld development set is the evaluation benchmark. Training tasks for reinforcement learning come from families that do not appear in this set. Family-level grouping captures the dependence between task variants.