EVOPOLICYGYM: BENCHMARKING CODING AGENTS FOR AUTONOMOUS POLICY EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-evolving agents that learn from interaction can improve their behavior by iteratively developing policies from environmental feedback. However, evaluating this capability remains challenging because agents may differ not only in the policies they produce, but also in how they explore the environment, allocate interaction budgets, and revise their policies over time. We introduce EvoPolicyGym, a benchmark framework for evaluating autonomous policy evolution under a standardized interaction budget. EvoPolicyGym assesses policy development trajectories and applies a validation protocol to select a policy from each run for independent evaluation on held-out episodes. The framework integrates 357 environment variants across 18 ecosystems. We evaluate five coding agents on eight selected tasks spanning games and continuous-control domains. The results show that Astra achieves the highest mean score on seven of eight evaluated tasks. Analysis of policy development trajectories further reveals that Astra iteratively identifies policy failures and incorporates the resulting evidence into subsequent policy revisions. These results establish EvoPolicyGym as a framework for studying how coding agents transform interaction experience into reusable executable policies, and provide evidence of emerging capabilities in autonomous policy development. Our anonymized code is available here.

1 INTRODUCTION

Autonomous agents are expected not merely to produce a strong initial solution, but to improve their behavior through interaction and feedback. Modern coding agents can invoke tools, inspect execution failures, and iteratively revise executable artifacts over long horizons (Yang et al., 2024; Wang et al., 2025; 2024). Similarly, iterative refinement and program evolution methods use feedback to improve outputs across multiple attempts (Shinn et al., 2023; Madaan et al., 2023; Novikov et al., 2025). However, evaluating this capability remains challenging. Final-task performance alone does not reveal *how* an agent improves: the same outcome may follow systematic diagnosis and adaptation or repeated trial and error (Sahoo et al., 2026; Majgaonkar et al., 2025). Repeated adaptation to observed evaluation feedback also raises the risk of overfitting (Dwork et al., 2015).

This raises a fundamental evaluation question: *how should we measure an agent's ability to autonomously improve a policy from interaction with an environment?* Existing benchmarks study agents within repository-level software engineering (Jimenez et al., 2024) and domain-specific engineering optimization (Chi et al., 2026), where performance also depends on navigating codebases and applying domain knowledge. These settings motivate a complementary evaluation focused on how effectively agents learn from environment feedback. Such a setting should control the amount of environment interaction available to each agent and preserve the full sequence of decisions through which policy improvement occurs.

We formalize this setting as *Autonomous Policy Evolution*: an agent repeatedly modifies an executable decision policy based on feedback from previous deployments, subject to a fixed interaction budget. The resulting object of evaluation is not only the final policy, but the entire sequence of policy revisions and associated training feedback. Policy quality is measured on held-out rollouts, allowing us to distinguish genuine improvements that generalize from changes that merely exploit observed feedback.

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.

To study Autonomous Policy Evolution systematically, we introduce EVOPOLICYGYM, a controlled benchmark built from compact interactive environments. In each task, an agent edits an executable policy system, submits it for platform-controlled evaluation, observes server-generated rollout feedback, and decides how to revise the policy under a fixed budget. Building on work on executable policy programs and reusable skills (Liang et al., 2022; Wang et al., 2023), EVOPOLICYGYM makes their development under matched interaction budgets an object of evaluation. By recording the complete execution–feedback–revision trajectory, the benchmark supports analysis not only of final held-out performance, but also of when improvements arise, whether they persist, how interaction budgets are allocated, and how agents use feedback to diagnose and correct policy failures.

EVOPOLICYGYM provides a shared interface to **357 named environment variants across 18 ecosystems** (Figure 2). We evaluate five coding agents on eight tasks spanning games, planning, visual control, and manipulation. For each model–task pair, we conduct three independent runs with a budget of 128 training episodes, yielding 120 runs, supplemented by 24 Astra runs with minimal feedback. We use these experiments to study four questions: **(RQ1)** How effective are the policies that agents ultimately develop? **(RQ2)** How does policy quality evolve over the course of autonomous development? **(RQ3)** Does richer environment feedback improve policy development? **(RQ4)** How do agents translate environment feedback into policy improvements?

Our results reveal differences in both the quality and the dynamics of autonomous policy development, and highlight the role of environment feedback. Astra achieves the highest mean final score on seven of the eight evaluated tasks (Table 2). Held-out development trajectories show that agents differ in when improvements emerge, whether gains persist across revisions, and how much they benefit from additional interaction (Figure 3). A feedback ablation finds that richer environment feedback yields higher mean final scores for Astra on all eight tasks, with benefits varying considerably by task (Figure 4). Qualitative case studies show how Astra translates feedback into concrete policy improvements through failure diagnosis, revisions to decision rules, and parameter search (Figure 5). Together, these results provide a controlled view of how coding agents improve executable policies through interaction. More broadly, EVOPOLICYGYM offers a way to study not only what agents ultimately achieve, but how effectively they turn feedback into sustained policy improvements.

2 RELATED WORK

Evaluating autonomous self-improvement for agents. A line of research has shown that language-model-based systems can improve executable skills, reward functions, and algorithms through iterative feedback (Wang et al., 2023; Ma et al., 2023; Romera-Paredes et al., 2024; Novikov et al., 2025). A complementary line of interactive benchmarks places agents in web, operating-system, database, and workplace environments to assess tool use, state tracking, and multi-step decision making (Liu et al., 2023; Zhou et al., 2024; Xie et al., 2024; Drouin et al., 2024). Their primary evaluation unit is task completion within an interaction episode, rather than the development of a persistent executable policy across repeated deployments. A related line of experimentation benchmarks studies iterative machine-learning system development, where agents design, execute, and refine models or training pipelines under repeated feedback (Huang et al., 2024; Chan et al., 2024). While these settings introduce longer optimization loops, they focus on machine-learning experiments and pipelines. Our work evaluates *harness–model agents* as developers of executable decision policies, using bounded environment interaction and a shared evaluation interface across tasks.

Bounded optimization and trajectory-level analysis for agents. Frontier-Eng studies generative engineering optimization, where agents improve executable artifacts using verifier feedback under a fixed interaction budget, and analyzes how gains evolve during search (Chi et al., 2026). It covers a broad class of engineering problems, while our focus is the development of decision policies for interactive environments through episode-budgeted evaluation. In this setting, feedback is mediated through a platform-controlled protocol, and saved policy revisions can be evaluated on held-out episodes. Prior trajectory analyses show that aggregate success rates can obscure retries, regressions, and missing verification (Sahoo et al., 2026; Majgaonkar et al., 2025). Studies of real developer–agent sessions further document user corrections, discarded code, and failures to follow developer intent (Baumann et al., 2026; Tang et al., 2026). EvoPolicyGym connects such analysis of development behavior to the measured performance of successive policy revisions under matched interaction budgets.

3 EvoPolicyGym: Task and Evaluation Protocol

EvoPolicyGym evaluates a coding agent's ability to improve an executable policy through environment interaction. Starting from an initial policy package, the agent develops and tests revisions under a fixed episode budget. The resulting policy is evaluated on held-out instances of the same task. Figure 1 outlines this process.

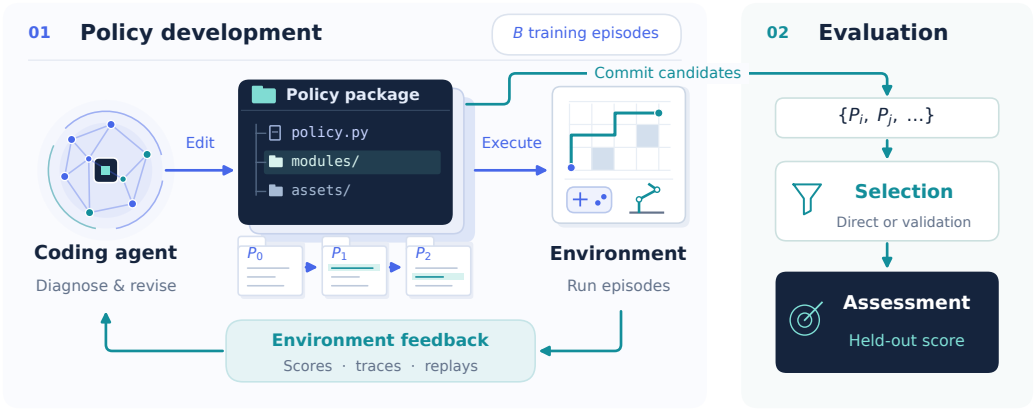


Figure 1: **EvoPolicyGym evaluation protocol.** The agent edits a policy package and uses environment feedback to guide revisions. Each P_i is a frozen package snapshot. After development, selection takes either the agent's single candidate or the winner of optional Validation. Assessment measures the selected policy's held-out performance.

3.1 Task formulation

The agent receives a task description, an initial policy program P_0, and a budget of B training episodes. A **program** is a policy package containing an executable entry point and any supporting modules or local assets. The agent may revise its structure, decision rules, planning routines, and numerical parameters. The **environment** defines the observations, legal actions, termination conditions, and scoring rule. The objective is to produce a policy that performs well on new instances of this configured task.

The coding agent develops the program; the policy executes it, mapping observations to actions without invoking the coding agent at each step. An **episode** is one environment trial, starting with a fresh policy instance. Policy state may persist within an episode, while knowledge retained across episodes is encoded in the program. A **run** comprises the complete development and evaluation process. The coding agent's configuration remains fixed during a run; its workspace accumulates code, feedback, and analysis as development proceeds. Appendix B.1 specifies the policy interface.

3.2 Feedback-driven policy development

Development alternates between editing the policy package and testing it in the environment. For each experiment, the agent chooses a batch from a fixed pool of indexed training inputs. Reusing indices allows it to compare revisions on the same inputs; choosing new indices tests behavior on additional instances. The underlying scenario specifications and seeds remain private.

Each admitted request consumes its full episode allocation before execution. Repeated trials consume budget again, so the agent decides how to distribute its allowance across revisions and test batches. Batch limits and completion rules are fixed for each run (Appendix B.2).

Each experiment executes a frozen snapshot of the policy package. A published **submission** records this snapshot together with the training indices, budget charge, and resulting feedback. Feedback contains scores and task-provided information such as episode summaries, diagnostic traces, or replays; its content is determined by the task configuration. The agent can inspect these records, write diagnostic tools, and use its findings to revise the package. It may continue from an earlier version,

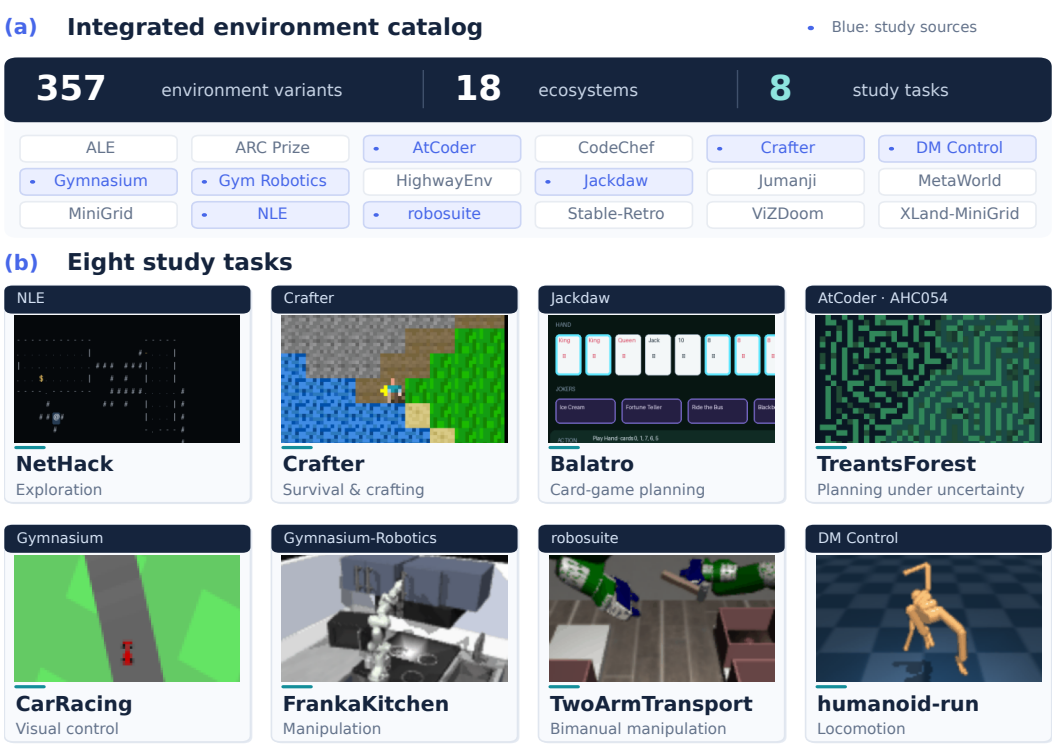


Figure 2: **Benchmark coverage and study tasks.** (a) The catalog contains 357 environment variants from 18 ecosystems; blue labels identify sources used in this study. (b) The eight study tasks, illustrated with replay images. Images depict the environments and may differ from policy observations.

compare alternative approaches, or retain several candidates. Saved snapshots preserve the exact programs associated with these experiments for subsequent selection and evaluation.

3.3 Policy selection and held-out assessment

Development ends when the agent commits its candidate submissions. The protocol supports two selection modes: the agent submits a single candidate for direct selection, or commits a shortlist for **Validation**. When enabled, Validation evaluates the committed candidates on matched trials after the agent exits and selects P^* using the benchmark's primary score. The selection mode and candidate limit are fixed before development; tie-breaking rules appear in Appendix B.3.

Assessment evaluates P^* on a separate held-out allocation and supplies the run's final score. Validation and Assessment use allocations outside the development budget and return no feedback to the agent. The selected program remains fixed throughout Assessment. Comparisons within a task use matched initial programs, task configurations, and evaluation settings, with any deliberate changes to the development conditions specified by the experiment. Detailed failure handling and incomplete-run outcomes are given in Appendix B.4.

4 Benchmark Suite

EvoPolicyGym integrates **357 named environment variants across 18 ecosystems** in 64 independently installable distributions (Figure 2). Variants denote implemented configurations, excluding episode seeds and generated instances; they are not independent task families. Appendix A.4 details the catalog.

We select eight tasks spanning exploration, planning, visual control, manipulation, and locomotion (Table 1). NetHack (Küttler et al., 2020) and Crafter (Hafner, 2021) require exploration and resource

management; Balatro (TylerFlar, n.d.) and TreantsForest (AtCoder, 2025) test planning with incomplete information. CarRacing (Towers et al., 2024) couples visual perception with continuous control, while FrankaKitchen (Farama Foundation, n.d.), TwoArmTransport (Zhu et al., 2025), and humanoid-run (Tassa et al., 2018) test manipulation and locomotion from numeric state. Together, they cover discrete, structured, and continuous actions, with visual or structured observations.

Table 1: **Observation spaces, action spaces, and scoring metrics for the eight study tasks.** Observations and actions refer to the policy interface. Higher scores are better on every task; episode returns are averaged over the evaluation episodes.

Task	Observation	Action	Primary score
NetHack	Map, status, text	Discrete	Shaped return
Crafter	RGB image	Discrete	Achievement score (%)
Balatro	Game state	Structured	Mean run score
FrankaKitchen	Numeric state	Continuous	Episode return
CarRacing	RGB image	Continuous	Episode return
TreantsForest	Map and reveal updates	Placements	Mean capped movement count
TwoArmTransport	Numeric state	Continuous	Shaped return
humanoid-run	Numeric state	Continuous	Episode return

Each task provides scores and feedback such as achievement breakdowns, state traces, or replays to help agents inspect policy behavior. Appendix A.1 describes the tasks, scoring rules, and evaluation settings.

5 Experiments

Our experiments examine both the policies agents produce and the process by which they develop them. We first compare final policy performance across five agents on eight tasks (RQ1), then trace how performance changes as agents use their interaction budget (RQ2). To assess the role of environment feedback, we compare Astra under rich and minimal feedback (RQ3). Finally, we inspect Astra's development logs to understand how it uses feedback to diagnose errors, revise decision rules, and search policy parameters (RQ4).

5.1 Setup

We evaluate Luna (`gpt-5.6-luna`), Terra (`gpt-5.6-terra`), Sol (`gpt-5.6-sol`), Astra (`gpt-6-astra`), and DeepSeek V4.1 Flash (`deepseek-v4.1-flash`) using the Codex 0.154.0 harness with `xhigh` reasoning. Each model is evaluated on all eight tasks with three independent agent sessions and separate workspaces per task, giving 120 completed runs. RQ3 adds 24 Astra runs with minimal feedback, for 144 runs in total.

Each run must exhaust a budget of 128 training episodes. All models and repeats use the same initial Program and environment-seed schedule within each task. Validation evaluates up to three candidates on 16 episodes each; Assessment evaluates the selected policy on 32 test episodes. Appendix A.1 lists task-specific settings and implementation limits.

5.2 RQ1: How well do agents develop policies?

Table 2 reports final policy scores for the five agents, with the initial policy P_0 included as a reference. Astra has the highest mean on seven of eight tasks. On NetHack, it scores 844.77 compared with 235.31 for the next-best agent, DeepSeek. On Crafter, it reaches 63.95% compared with Sol's 24.98%. DeepSeek exceeds Luna on all eight tasks, Terra on five, and Sol on three. It has the highest mean on Humanoid (1.127 versus Astra's 1.122), although the difference is small relative to the variation across runs (SDs of 0.441 and 0.052, respectively).

Score differences between agents vary substantially by task. On TwoArmTransport, Astra scores 97.29, while DeepSeek and Sol reach only 2.06 and 1.73, and Luna and Terra score zero. CarRacing shows a more gradual increase across Luna, Terra, Sol, and Astra: 213.04, 403.53, 546.31, and

Table 2: **Final policy performance after 128 training episodes.** P_0 is the shared initial policy; agent columns report final policies selected by Validation after 128 training episodes. Figure 3 tracks performance across successive policy revisions from P_0. Each policy is evaluated on 32 held-out episodes. Agent results are means ± SD across three runs (SD in superscript); P_0 is evaluated once. Bold marks the highest mean per task.

	Ref.	Developed policies $B = 128$				
Environment	P_0	**Luna**	**Terra**	**Sol**	**Astra**	**DeepSeek**
NetHack	36.59	$98.93^{\pm 55.11}$	$126.93^{\pm 61.11}$	$190.63^{\pm 40.05}$	$\mathbf{844.77}^{\pm 222.96}$	$235.31^{\pm 156.94}$
Crafter	1.06	$4.05^{\pm 0.95}$	$7.43^{\pm 0.91}$	$24.98^{\pm 5.20}$	$\mathbf{63.95}^{\pm 8.47}$	$20.13^{\pm 3.83}$
Balatro	2.75	$5.90^{\pm 0.31}$	$8.12^{\pm 1.30}$	$11.31^{\pm 1.67}$	$\mathbf{20.26}^{\pm 2.46}$	$8.51^{\pm 0.73}$
FrankaKitchen	0.00	$0.41^{\pm 0.53}$	$1.00^{\pm 0.00}$	$1.33^{\pm 0.58}$	$\mathbf{3.99}^{\pm 0.98}$	$0.67^{\pm 0.58}$
CarRacing	−26.14	$213.04^{\pm 159.04}$	$403.53^{\pm 162.23}$	$546.31^{\pm 136.80}$	$\mathbf{918.09}^{\pm 9.25}$	$313.94^{\pm 285.71}$
TreantsForest	98.72	$160.26^{\pm 37.05}$	$645.79^{\pm 413.71}$	$721.36^{\pm 331.36}$	$\mathbf{1318.74}^{\pm 205.67}$	$464.00^{\pm 128.44}$
TwoArmTransport	0.00	$0.00^{\pm 0.00}$	$0.00^{\pm 0.00}$	$1.73^{\pm 3.00}$	$\mathbf{97.29}^{\pm 4.53}$	$2.06^{\pm 3.57}$
Humanoid	0.88	$0.99^{\pm 0.09}$	$0.93^{\pm 0.02}$	$0.97^{\pm 0.08}$	$1.12^{\pm 0.05}$	$\mathbf{1.13}^{\pm 0.44}$

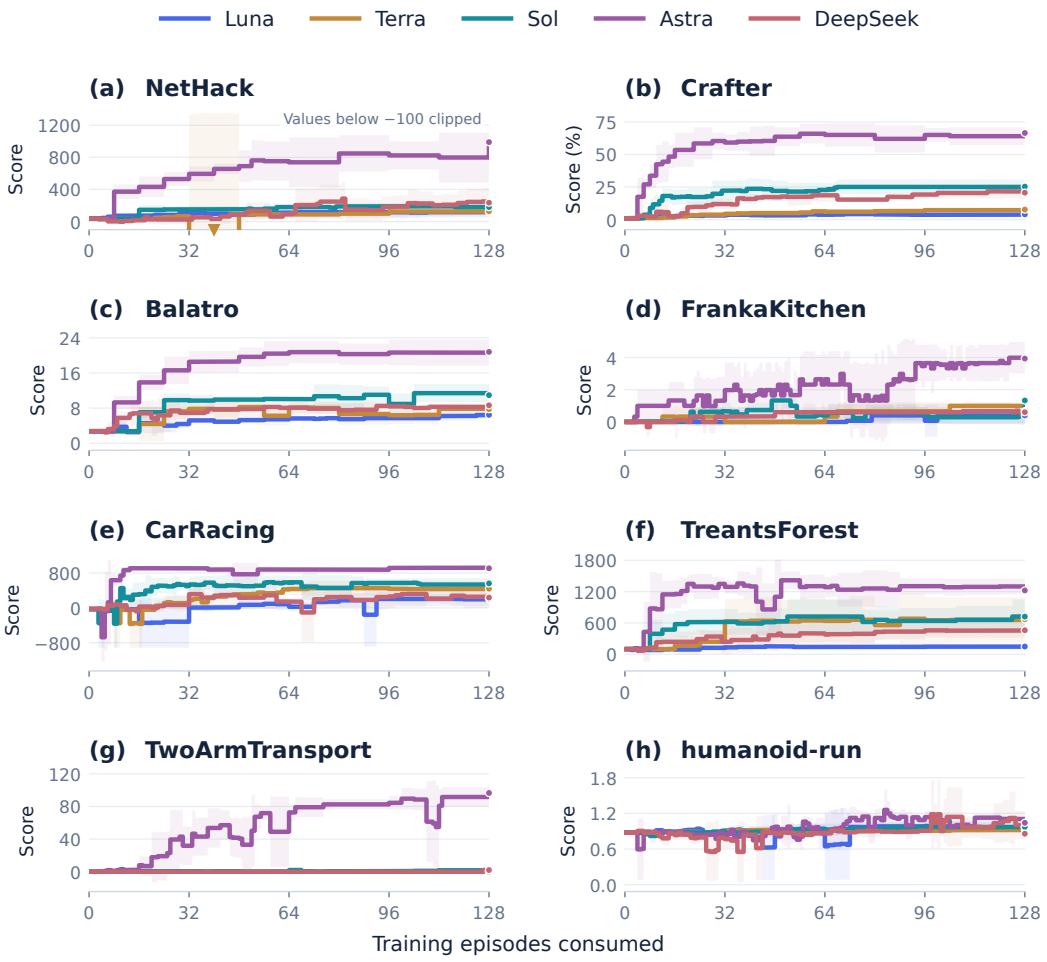


Figure 3: **Policy performance during development.** Held-out performance generally improves as agents iteratively revise their policies using environment feedback, although individual revisions can cause regressions. Lines show mean scores across three independent runs, with each run's latest submitted policy evaluated offline on 32 held-out episodes; shading indicates ±1 standard deviation. NetHack scores below −100 are clipped; the triangle marks the clipped segment.

918.09. On Humanoid, however, all five means fall between 0.93 and 1.13, near the initial-policy reference of 0.88.

Variation across runs is also substantial in several conditions. On CarRacing, Astra's SD is 9.25, compared with 136.80–285.71 for the other agents. On TreantsForest, Terra and Sol have SDs of 413.71 and 331.36. Consistency alone does not imply a strong policy: Terra scores 1.00 in every FrankaKitchen run, below Sol's mean of 1.33 and Astra's 3.99.

5.3 RQ2: How does performance change with interaction budget?

We analyze the policy optimization process by tracking how agents iteratively refine their policies through environment feedback and how policy performance changes across successive revisions.

Figure 3 tracks the latest submitted policy throughout all 120 runs, with saved policies evaluated offline on 32 held-out episodes. To distinguish the first revision from subsequent refinement, we compare the supplied program P_0, the first agent-revised submitted policy P_1, and the latest submitted policy P_{last} after 128 training episodes. Across three Astra runs per task, the mean scores at these stages are $36.59 \rightarrow 372.27 \rightarrow 989.00$ on NetHack, $1.06\% \rightarrow 27.05\% \rightarrow 66.46\%$ on Crafter, $2.75 \rightarrow 9.30 \rightarrow 20.79$ on Balatro, $0.00 \rightarrow 0.00 \rightarrow 3.94$ on FrankaKitchen, $-26.14 \rightarrow -1000.00 \rightarrow 909.61$ on CarRacing, $98.72 \rightarrow 1163.83 \rightarrow 1222.53$ on TreantsForest, $0.00 \rightarrow 0.00 \rightarrow 96.51$ on TwoArmTransport, and $0.876 \rightarrow 0.594 \rightarrow 1.041$ on Humanoid. CarRacing's P_1 incurs the policy-failure score on all 96 held-out episodes, so its later improvement includes recovery from execution failures. TreantsForest gains mainly in the first revision, whereas FrankaKitchen and TwoArmTransport obtain positive returns only in later revisions; Humanoid remains at low returns.

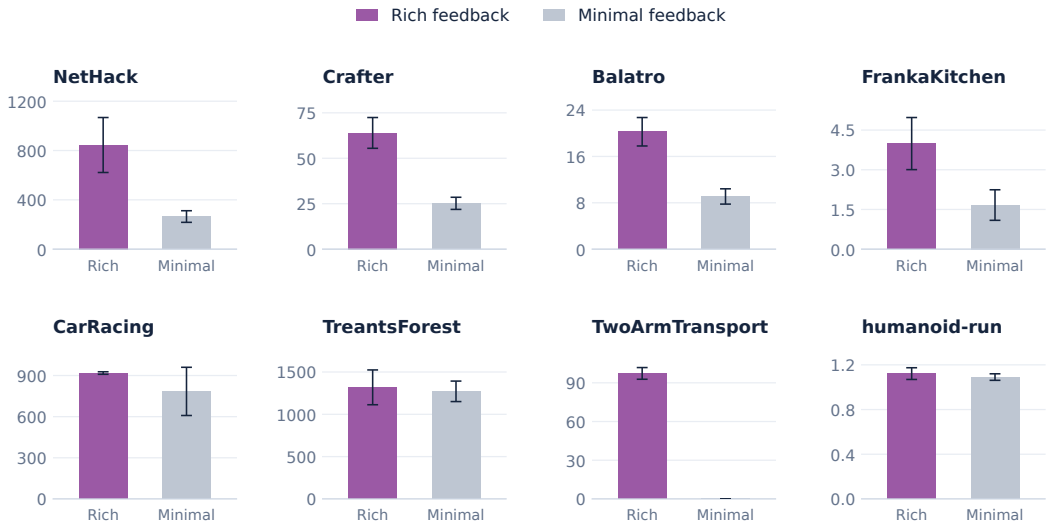


Figure 4: **Effect of environment feedback on Astra's final policy performance.** Bars show mean scores across three independent runs per condition; error bars indicate standard deviations. Each selected policy is assessed on 32 held-out episodes. Rich feedback includes environment diagnostics and replay artifacts; minimal feedback retains scores, execution status, and budget information. Each panel uses the task's scoring scale (Crafter: achievement %); higher is better. Rich-feedback scores are the Astra results in Table 2. Table 7 reports the exact values.

5.4 RQ3: Does richer environment feedback improve policy development?

We compare Astra under rich and minimal feedback on all eight tasks, with three independent runs per task in each condition. *Rich feedback* includes scores, environment diagnostics, and task-specific trajectories, observations, or visual replays. *Minimal feedback* includes the aggregate score, per-episode rewards, execution status and failure codes, and budget information, but omits step counts, diagnostics, and feedback artifacts. Policy observations during execution are identical in both conditions.

Rich feedback yields a higher mean score on every task (Figure 4). With minimal feedback, NetHack falls from 844.77 to 264.60 and Crafter from 63.95% to 25.21%; Balatro and FrankaKitchen also lose more than half their mean score. The clearest contrast is TwoArmTransport: rich feedback achieves 97.29, whereas all three minimal-feedback runs score zero, without policy execution failures during Assessment. The differences on TreantsForest and Humanoid are much smaller. Across these tasks, richer environment feedback helps Astra develop better policies, with the size of the benefit varying considerably by task.

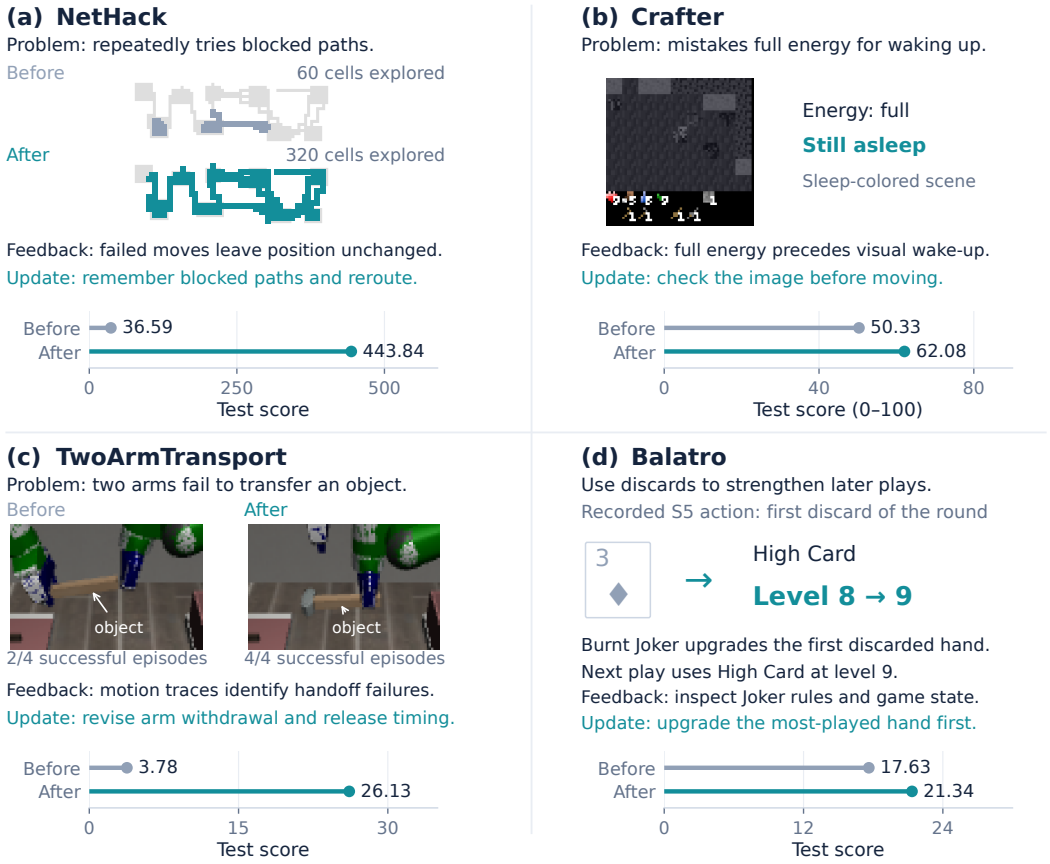


Figure 5: **Feedback-guided revisions in four Astra runs.** (a) NetHack: remembering blocked paths. (b) Crafter: detecting sleep from images. (c) TwoArmTransport: revising withdrawal and release timing. (d) Balatro: investing discards in hand upgrades. Scores before and after each revision are measured on 32 held-out episodes using each task's scoring scale. Revisions may contain multiple edits, so score changes cannot be attributed to individual edits. Appendix D provides the underlying traces and code changes.

5.5 RQ4: How do agents use feedback to improve policies?

We inspect Astra's 24 rich-feedback runs to trace how observations lead to code changes. Figure 5 shows four cases; Appendix D documents the commands, revisions, and measurements. This analysis describes Astra's behavior within these runs; it does not explain performance differences between models.

Diagnosing state errors. In NetHack, Astra writes diagnostic programs to inspect message frequencies and position traces. After identifying repeated blocked moves, it adds blocked-edge memory and route/door handling in S2. In Crafter, replaying the policy on recorded observations exposes map drift around waking: the energy indicator reaches its maximum before sleep ends. S5 adds an image-based sleep detector along with perception and planning changes. In both cases, inspecting traces reveals a specific error that the aggregate score alone does not identify.

Repairing handoff coordination. In TwoArmTransport, Astra uses end-effector and object traces to diagnose failures during handoff and arm separation. In Run 1, the S9–S10 revision raises the donor arm's withdrawal target, removes a simultaneous lift of the receiving arm, and advances release from step 480 to 465. These changes address coordination between grasping, withdrawal, and placement: the donor must clear the object while the receiving grasp remains stable. Success increases from two to four of the same four training episodes, and the score on 32 held-out episodes rises from 3.78 to 26.13 (Figure 5c). The case illustrates how feedback guides revisions to transitions between policy stages.

Investing discards in hand upgrades. In Balatro, Astra inspects Joker rules and adds a Burnt Joker strategy: upgrade its most-played hand type on each round's first discard. A recorded S5 action raises High Card from level 8 to 9 before the next play. S4–S5 scores rise from 18.88 to 23.75 on eight matched training episodes and from 17.63 to 21.34 on 32 held-out episodes. These gains cover a bundled revision that also changes discard planning and purchase values.

Searching policy parameters. In FrankaKitchen, Astra writes a search script that perturbs terminal joint targets and ranks trials by return plus a kettle-distance term. S61 and S71 differ in just one terminal joint target; on the same training episode, the number of completed tasks rises from four to five. Feedback here guides a parameter search within an existing policy.

Successful revisions coexist with frequent regressions: 23 of the 24 Astra runs include a later version that scores lower on matched training indices. Appendix D also examines an unsuccessful humanoid revision. The selected cases show how feedback can lead to useful changes, but such changes do not reliably improve every policy.

6 LIMITATIONS

Our experiments use a single budget configuration: 128 training episodes, 16 validation episodes per candidate, and 32 assessment episodes per selected policy. Although these budgets are independently configurable in EVOPOLICYGYM, our results do not establish how policy quality or agent rankings change with additional interaction. Larger training budgets may enable further exploration and refinement, while more validation and assessment episodes may improve selection reliability and estimation precision.

Task difficulty also affects the benchmark's ability to distinguish agents. Under the current budget and initial policies, some tasks remain beyond the demonstrated capabilities of the evaluated agents. On humanoid-run, all five agents obtain similarly low returns, and the inspected Astra trajectories do not exhibit sustained running. This floor effect limits discrimination and makes small numerical gains difficult to interpret as meaningful behavioral improvements. Future work should jointly examine task difficulty, interaction budgets, and initial policies, and complement episode returns with behavioral milestones such as stable standing and sustained locomotion.

7 CONCLUSION

We introduced EVOPOLICYGYM to study autonomous policy development using controlled interaction budgets, recorded revisions, and held-out assessment. Astra achieves the highest mean final score on seven of eight tasks. Across all eight tasks, Astra obtains higher mean final scores with rich feedback than with minimal feedback. Development trajectories reveal uneven progress and regressions, while case studies link feedback to failure diagnosis, rule revision, and parameter search. Together, these results illustrate the utility of EvoPolicyGym as an evaluation protocol and framework for feedback-driven policy development.

AI USE STATEMENT

Generative AI assistance was used in discussing the research framing, identifying initial references, drafting manuscript text, organizing the experiment plan, creating the editable overview diagram, and preparing the LaTeX source, experiment configuration and result tables, and illustrative graphics retained separately.

REFERENCES

AtCoder. AHC054: A—Treant's Forest. Official task statement, 2025. URL https://atcoder.jp/contests/ahc054/tasks/ahc054_a. Accessed September 13, 2026.

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. SWE-chat: Coding agent interactions from real users in the wild, 2026. URL https://arxiv.org/abs/2604.20779. arXiv:2604.20779.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: Evaluating machine learning agents on machine learning engineering, 2024. URL https://arxiv.org/abs/2410.07095. arXiv:2410.07095.

Yizhe Chi, Deyao Hong, Dapeng Jiang, Tianwei Luo, Kaisen Yang, Boshi Zhang, Zhe Cao, Xiaoyan Fan, Bingxiang He, Han Hao, Weiyang Jin, Dianqiao Lei, Qingle Liu, Houde Qian, Bowen Wang, Situ Wang, Youjie Zheng, Yifan Zhou, Calvin Xiao, Eren Cai, and Qinhuai Na. Frontier-Eng: Benchmarking self-evolving agents on real-world engineering tasks with generative optimization, 2026. URL https://arxiv.org/abs/2604.12290v2. arXiv:2604.12290v2.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11642–11662. PMLR, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015. doi: 10.1126/science.aaa9375. URL https://doi.org/10.1126/science.aaa9375.

Farama Foundation. Franka kitchen: Gymnasium-Robotics documentation. Official environment documentation, n.d. URL https://robotics.farama.org/envs/franka_kitchen/franka_kitchen/. Accessed September 13, 2026.

Danijar Hafner. Benchmarking the spectrum of agent capabilities, 2021. URL https://arxiv.org/abs/2109.06780. arXiv:2109.06780.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 20271–20309. PMLR, 2024. URL https://proceedings.mlr.press/v235/huang24y.html.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack learning environment, 2020. URL https://arxiv.org/abs/2006.13760. arXiv:2006.13760.

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, 2022. URL https://arxiv.org/abs/2209.07753. arXiv:2209.07753.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents, 2023. URL https://arxiv.org/abs/2308.03688. arXiv:2308.03688.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models, 2023. URL https://arxiv.org/abs/2310.12931. arXiv:2310.12931.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2019. URL https://arxiv.org/abs/2303.17651.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories, 2025. URL https://arxiv.org/abs/2511.00197. arXiv:2511.00197.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131. arXiv:2506.13131.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Priyam Sahoo, Gaurav Mittal, Xiaomin Li, Shengjie Ma, Benjamin Steenhoek, Pingping Lin, and Yu Hu. AgentLens: Revealing the lucky pass problem in SWE-Agent evaluation, 2026. URL https://arxiv.org/abs/2605.12925v3. arXiv:2605.12925v3.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-0377. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Ningzhi Tang, Chaoran Chen, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, Tao Dong, and Toby Jia-Jun Li. How coding agents fail their users: A large-scale analysis of developer-agent misalignment in 20,574 real-world sessions, 2026. URL https://arxiv.org/abs/2605.29442. arXiv:2605.29442.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind control suite, 2018. URL https://arxiv.org/abs/1801.00690. arXiv:1801.00690.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032. arXiv:2407.17032.

TylerFlar. jackdaw-balatro: Source repository. Source repository, n.d. URL https://github.com/TylerFlar/jackdaw-balatro. Accessed September 13, 2026.

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

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning*

Research, pp. 50208–50232. PMLR, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2407.16741.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://arxiv.org/abs/2404.07972.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1601. URL https://arxiv.org/abs/2405.15793.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

Yuke Zhu, Josiah Wong, Ajay Mandlekar, Roberto Martín-Martín, Abhishek Joshi, Kevin Lin, Abhiram Maddukuri, Soroush Nasiriany, and Yifeng Zhu. robosuite: A modular simulation framework and benchmark for robot learning, 2025. URL https://arxiv.org/abs/2009.12293v3. arXiv:2009.12293v3.

A Study Tasks and Environment Coverage

The eight study tasks use the policy interfaces and primary metrics in Table 1. This appendix specifies their starting policies, scoring profiles, and development feedback, followed by the full catalog.

A.1 Task profiles and initial policies

The descriptions below follow the recorded task configurations and saved initial programs. Each task has a common starting package across models and feedback conditions.

NetHack. The agent develops an exploration and survival policy for `NetHackScore-v0`, using a semantic observation containing the map, status, messages, and inventory, with discrete NLE actions. The initial policy favors less-visited visible cells, searches periodically, and attempts to eat known food when hungry. The recorded profile uses a Monk character and a 5,000-step horizon. Its return includes the configured step penalty and therefore differs from raw game score.

Crafter. The task is to survive and unlock achievements from 64×64 RGB observations using 17 discrete actions. The initial package provides perception, memory, and capability interfaces with unfinished decision modules; its fallback alternates short movement sequences with interaction. Episodes end on death or at 10,000 steps. The study uses `achievement-score-v1`, the shifted geometric mean of success percentages across 22 achievements, rather than cumulative survival reward.

Balatro. The headless Red Deck, White Stake profile exposes game state and structured actions for playing hands, purchasing cards, and advancing rounds. The initial policy ranks candidate hands by poker category and card ranks, buys affordable Jokers when slots are available, and advances through the game phases. Under `run-score-v3`, an Episode scores N for a loss and $2N$ for a win, where N counts cleared Blinds. Skipped Blinds do not count; clearing 24 and winning scores 48. A Policy failure scores zero.

FrankaKitchen. The policy controls nine continuous action components from numeric robot and goal observations to complete kitchen manipulation tasks. The starting policy returns a zero-action vector. Each step rewards newly completed tasks, and the primary score is mean Episode return. Completion and goal-distance diagnostics provide additional information about progress.

CarRacing. The policy drives from RGB observations using continuous steering, acceleration, and braking. The initial policy always outputs $[0, 0.5, 0]$, applying constant throttle without steering or braking. The study uses `CarRacing-v3` with domain randomization disabled and a lap-completion threshold of 0.95. The primary score is mean Episode return, reflecting track coverage and the environment's penalties.

TreantsForest. The policy places trees to delay an adventurer reaching a flower, using the public map and reveal updates. The initial policy places no trees. The independent rule implementation generates grids of size 20–40; each accepted turn earns one point. An Episode ends when the adventurer reaches the flower or at the 2,048-turn cap. The score is mean capped movement count, so larger values indicate longer delays.

TwoArmTransport. Two Panda arms must transfer an object using 14 continuous action components and proprioceptive and object-state observations. The recorded profile uses opposed arms with the `BASIC/OSC_POSE` controller, shaped rewards, and a 500-step horizon. The initial policy outputs zero actions. Mean Episode return is the primary score; success is reported as an auxiliary outcome.

Humanoid. The policy controls 21 action components from the named numeric state observations of DM Control's `humanoid-run` task. The initial package decodes the observation tensors and returns zero actions. Episodes use a 1,000-step horizon, and the primary score is mean cumulative return.

A.2 Environment feedback

Both RQ3 conditions retain the aggregate score, individual Episode rewards, status and failure codes, and submission and budget metadata. Rich feedback additionally provides the task-specific information in Table 3. Minimal feedback omits these diagnostics, step counts, and artifacts. The policy's observation and action interface is unchanged. Camera images in the numeric-state tasks are available to the coding agent during development.

Table 3: **Additional environment feedback in the rich condition.** Both conditions provide scores and Episode outcomes. Entries list the task-specific diagnostics and artifacts omitted in the minimal condition. Traces and image sequences follow each task's capture and retention limits.

Task	Diagnostic information	Recorded artifacts
NetHack	Depth, game score, deaths, and stalled-step statistics	Semantic observations and action/message trajectories
Crafter	Achievement success rates and action statistics	RGB observations and trajectories
Balatro	Hand settlements, Blind progress, purchases, and resources	Structured game replay
FrankaKitchen	Task completion, goal progress, and action/state statistics	State/action traces and camera frames
CarRacing	Track-coverage estimates and control summaries	Observation frames, sampled traces, and replays
TreantsForest	Path lengths, reveal coverage, and placement statistics	Turn and placement traces
TwoArmTransport	Success, reward, and object/robot motion	State/action traces and camera frames
Humanoid	Reward, action magnitude, and state motion	State/action traces and camera frames

A.3 Shared evaluation settings

All conditions use environment seed 20260914, a 128-Episode training allowance and a pool of 128 indexed inputs. Development permits at most 128 submissions and 32 Episodes per submission, with budget exhaustion required for completion. Episode and Agent-session time limits are 120 seconds and six hours, respectively; bulk Feedback retention is limited to 1 GiB. The configured harness is Codex 0.154.0 with `xhigh` reasoning and no additional Agent Skills.

Table 4 summarizes the evaluation allocations and Episode limits used in the study.

Table 4: **Evaluation settings for the eight study tasks.** Validation lists episodes per candidate and the maximum number of candidates. Every task uses a training budget of 128 episodes.

Environment	**Train pool**	**Max batch**	**Validation**	**Assessment**	**Episode limit (s)**
NetHack	128	32	$16 \times \leq 3$	32	120
Crafter	128	32	$16 \times \leq 3$	32	120
Balatro	128	32	$16 \times \leq 3$	32	120
FrankaKitchen	128	32	$16 \times \leq 3$	32	120
CarRacing	128	32	$16 \times \leq 3$	32	120
TreantsForest	128	32	$16 \times \leq 3$	32	120
TwoArmTransport	128	32	$16 \times \leq 3$	32	120
Humanoid	128	32	$16 \times \leq 3$	32	120

A.4 FULL ENVIRONMENT CATALOG

The 2026-09-13 source inventory contains 357 named environment variants across 18 source ecosystems and 64 independently installable distributions (Table 5). An entry is a distinct pair of source ecosystem and implemented environment/task identifier. Explicitly named size, difficulty, sensor, and version variants count separately; this is not a count of independent task families. For example, the MiniGrid total includes BabyAI, WFC presets, registered grid sizes, and supported versions.

We count the 25 pinned ARC-AGI-3 games and the 50 MetaWorld MT1 tasks individually. Their aggregate and custom collections add no entries. Episode seeds, generated instances, scoring variants, horizons, camera settings, and action-mode or other parameters retaining the same environment identifier do not increase the count. Deferred and unavailable integrations are excluded. Thus, the inventory does not enumerate every possible Benchmark configuration.

Table 5: **Environment catalog by source ecosystem.** Variant counts follow the identifier-based rule described above. Distributions are independently installable packages.

Ecosystem	Environment variants	Distributions
ALE	1	1
ARC Prize	25	1
AtCoder	3	3
CodeChef	1	1
Crafter	1	1
DM Control	28	1
Gymnasium	24	23
Gymnasium-Robotics	21	1
HighwayEnv	10	1
Jackdaw	1	1
Jumanji	18	1
MetaWorld	50	1
MiniGrid (including BabyAI and WFC)	136	23
NLE	1	1
robosuite	19	1
Stable-Retro	1	1
ViZDoom	12	1
XLand-MiniGrid	5	1
Total	357	64

The accompanying inventory script extracts named identifiers from the integrated adapters and configuration tables without importing simulators. The catalog describes implemented variants; the empirical study evaluates the eight tasks specified above.

B DETAILED PROTOCOL RULES

B.1 POLICY PACKAGE AND EXECUTION INTERFACE

A Program P implements a Policy π_P. At decision step ℓ, the Policy receives observation o_ℓ, internal state m_ℓ, and public context c, and returns action a_ℓ and updated state:

$$(a_\ell, m_{\ell+1}) = \pi_P(o_\ell, m_\ell; c). \tag{1}$$

Each Episode starts a fresh Environment and, in the current process backend, a fresh Policy process, instance, and scratch directory. State m_ℓ may persist within that Episode; information carried into later Episodes must be encoded in the submitted Program. Assessment therefore evaluates the captured artifact without relying on a previously warmed-up instance.

Public Policy context contains observation and action specifications, public metadata, environment parameters, and a dedicated Policy random seed. The PolicyValue ABI permits bounded scalars, typed tensors, and nested containers. It excludes custom simulator objects, private Case identifiers, Host paths, file descriptors, credentials, pool identity, and scoring objects. The Policy seed is distinct from

the Host-owned environment seed. The Agent's public training indices and the Policy's observation interface are separate information surfaces.

B.2 Experiment requests, budget, and feedback

Before Agent execution, the Host fixes $\mathcal{D}_{\text{train}} = ((\xi_i, \rho_i))_{i=0}^{N-1}$, where ξ_i is a private Episode specification and ρ_i a dedicated Policy seed. Both remain stable at each public index. A request supplies a nonempty, strictly increasing list of indices in $\{0, \ldots, N-1\}$; repeating an index in a later request creates and charges a fresh Episode.

For experiment j, let I_j be the requested indices and let $\mathcal{J}_{\text{charged}}$ contain requests that reached budget reservation. Given an allowance B, accounting satisfies

$$B_{\text{used}} = \sum_{j \in \mathcal{J}_{\text{charged}}} |I_j| \le B, \qquad B_{\text{remaining}} = B - B_{\text{used}}. \tag{2}$$

Pool size N controls the available inputs; B controls the total admitted Episode allocation. These need not be equal.

Let n be the requested batch size. The Host first checks index validity, resource limits, and successful Program capture. It then deducts n Episode units before evaluation. Table 6 distinguishes this reservation from successful publication. A fault partway through a batch can therefore leave fewer than n completed Episodes while retaining the full charge. The Run records retain both the charged allocation and available execution progress.

Table 6: **Training-budget accounting and feedback publication.** Published submissions may contain episodes that ended in policy failure. Faults in trusted components are not scored as policy outcomes.

Event	Charge	Outcome
Invalid indices, resource rejection, or failed Program capture	0	No Submission is published; the Agent may correct the request while authority remains open.
Completed batch, including any scored Policy failures	n	Publish the Program, accounting state, and public Feedback; further requests remain subject to the configured limits.
Trusted evaluation fault or mismatched result identity	n	End the Run without publishing the attempted Submission.
Feedback publication fault	n	End the Run without admitting the attempted Submission as a candidate.
Invalid finish request, including budget remaining under strict completion	0	Do not commit candidates or close authority; permit a valid retry.
Valid finish request	0	Commit the ordered candidate identifiers and permanently close submission and finish authority.

Completion and committed artifacts. The Episode allowance is an upper bound when early completion is enabled; the study instead requires budget exhaustion. Each published Submission binds a captured Program to its evaluation and budget charge. Later workspace edits do not alter that candidate. Candidates are identified by Submission, so two candidates may contain identical Program content.

Feedback availability. The configured retention limit can prune older bulk artifacts while retaining compact results and the newest Submission's complete feedback. Agent-created analysis remains available. The study's limits are listed in Appendix A.3.

B.3 Candidate commitment and selection

Let $C = (j_1, \ldots, j_K)$ be the committed, ordered shortlist of distinct published Submission identifiers, with K bounded by the configured candidate limit. Let P_j denote the Program captured in Submission

j. Validation evaluates only the committed candidates with the same evaluation configuration, Run-seed-derived validation seed, and Episode count. Let v_k be primary score, f_k the Policy-failed Episode count, and $d_{\mathcal{E}} \in \{+1, -1\}$ the maximization or minimization direction. Selection follows

$$k^* = \arg\min_{1 \le k \le K}{}^{\text{lex}}\big(-d_{\mathcal{E}} v_k,\ f_k,\ k\big), \qquad P^* = P_{j_{k^*}}. \tag{3}$$

Primary score dominates; failure count and then committed order break ties. Validation does not search all prior Submissions. Without Validation, the Agent must commit exactly one candidate for direct selection.

B.4 Final scores and incomplete Runs

Let $\mathcal{E}_\theta$ denote the Benchmark with fixed task version and configuration θ, and let $\mathcal{D}_{\text{assess}}$ be the held-out Assessment allocation. For the selected Program P^*, Run $\mathcal{R}$ has final score

$$Y_{\mathcal{R}} = S_{\mathcal{E}_\theta}(P^*; \mathcal{D}_{\text{assess}}), \tag{4}$$

where S includes the Benchmark's declared treatment of Policy failures.

A Run without valid selection or completed Assessment has no $Y_{\mathcal{R}}$ and retains its terminal outcome. The general API also allows Assessment to be omitted, in which case no held-out score is supplied. Invalid actions are never repaired or replaced: a Policy failure stops the Episode before another simulator transition and enters the declared scoring rule. Trusted Environment, Backend, and cleanup faults invalidate the evaluation instead. A Validation fault produces no selection; an Assessment fault preserves selection but yields no completed score. The coordinator does not retry or select a fallback automatically.

C Supplementary Results and Measurement

The study comprises 120 model-comparison Runs and 24 additional Astra minimal-feedback Runs. RQ1 and RQ2 cover the five-model comparison; RQ3 compares the two Astra feedback conditions, and RQ4 examines the 24 rich-feedback Astra Runs. Each model–task–feedback condition has three independent development sessions.

C.1 RQ1: score aggregation and initial-policy reference

We report the mean and sample standard deviation of the three independent Run scores. Each score aggregates the selected policy's Assessment Episodes using the task scorer; the development Runs are the independent repeats.

Table 2 reports a single 32-Episode evaluation of P_0 as a reference, rather than a paired estimate of improvement.

C.2 RQ2: development trajectories

The complete offline snapshot contains 2,506 unique Program evaluations (80,192 test Episodes), mapped to all 2,985 initial-Program and submission positions across 120 complete Run trajectories. Saved programs are evaluated on held-out Episodes using each task's Assessment configuration. Identical package snapshots under identical evaluation configurations share one measurement, which is mapped back to their original submission positions.

Let z_j be the offline held-out score of submission j and let u_j be cumulative training allocation after evaluating that submission. The reported latest-Program trajectory is

$$L(u) = z_{j(u)}, \qquad j(u) = \max\{j : u_j \le u\}, \quad 0 \le u \le 128, \tag{5}$$

with the initial Program indexed by $j = 0$, $u_0 = 0$. If $z_{j(u)}$ is missing, $L(u)$ is unreported, even if an earlier Program has a measured score. Right-continuous steps denote the same saved Program between submissions; they do not interpolate the performance of unmeasured programs. Aggregation requires measured values for all three Runs at a budget point. The curve follows the latest submission, including regressions. Repeated package snapshots share a measurement without changing their budget positions.

C.3 RQ3: FEEDBACK COMPARISON

Table 7 gives the numerical results underlying Figure 4. Astra's mean score is higher with rich feedback on all eight tasks, with the largest differences in NetHack, Crafter, Balatro, and TwoArmTransport. The differences in TreantsForest and Humanoid are small relative to the variation across Runs.

Table 7: **Astra's final scores under rich and minimal feedback.** Values are means ± standard deviations across three independent runs per condition; standard deviations are shown in superscript. Each selected policy is assessed on 32 held-out episodes. Scores use each task's scoring scale; higher is better. The difference is rich minus minimal in the task's native score units. Rich-feedback scores are the Astra results in Table 2.

Task	Rich feedback	Minimal feedback	Rich − minimal
NetHack	$844.77^{\pm 222.96}$	$264.60^{\pm 47.09}$	+580.17
Crafter	$63.95^{\pm 8.47}$	$25.21^{\pm 3.34}$	+38.74
Balatro	$20.26^{\pm 2.46}$	$9.09^{\pm 1.32}$	+11.17
FrankaKitchen	$3.99^{\pm 0.98}$	$1.67^{\pm 0.58}$	+2.32
CarRacing	$918.09^{\pm 9.25}$	$784.23^{\pm 175.78}$	+133.86
TreantsForest	$1318.74^{\pm 205.67}$	$1271.61^{\pm 121.02}$	+47.12
TwoArmTransport	$97.29^{\pm 4.53}$	$0.00^{\pm 0.00}$	+97.29
Humanoid	$1.122^{\pm 0.052}$	$1.091^{\pm 0.029}$	+0.031

Table 8 reports the individual Run scores behind these summaries. In NetHack, Crafter, Balatro, FrankaKitchen, and TwoArmTransport, every rich-feedback Run scores above every minimal-feedback Run. TreantsForest and Humanoid have overlapping score ranges. CarRacing shows a wider spread under minimal feedback, including one score of 583.00.

Table 8: **Individual Astra results under the two feedback conditions.** Each entry is the final Assessment score of one independent development Run. Run numbers identify repeats within a condition; columns do not imply paired agent trajectories. Each selected policy is evaluated on 32 Episodes.

	Rich feedback			Minimal feedback		
Task	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
NetHack	593.78	1019.90	920.62	226.74	249.72	317.33
Crafter	68.52	54.18	69.15	21.42	27.71	26.50
Balatro	20.38	17.75	22.66	10.53	7.94	8.81
FrankaKitchen	4.97	3.00	4.00	2.00	2.00	1.00
CarRacing	909.22	917.37	927.67	861.82	907.85	583.00
TreantsForest	1186.09	1214.47	1555.66	1132.69	1354.09	1328.06
TwoArmTransport	102.03	96.84	93.00	0.00	0.00	0.00
Humanoid	1.178	1.112	1.075	1.121	1.089	1.062

D ASTRA CASE STUDIES ACROSS EIGHT ENVIRONMENTS

We review all 24 Astra development logs (three Runs per environment), linking agent explanations to diagnostic commands, policy revisions, and evaluations. S denotes a submission within a Run. Comparisons use matched training Episodes unless stated otherwise; revisions may contain several edits. Figures combine recorded observations and trajectories with source values or explicitly labeled pseudocode. Outcome strips summarize the corresponding evaluations.

D.1 NETHACK: DISTINGUISHING FAILED ACTIONS FROM EXPLORATION

In Run 1, diagnostic code compares movement-error messages, positions, and game time. The initial policy spends 57.5% of its actions without advancing the game and reaches a mean maximum depth of 1.0 on four Episodes.

S2 records the previous position, target, action, and turn. A failed move with a recognized blocking message adds a directed edge to within-Episode memory; route planning then avoids that edge. Explicit door handling supplies another missing interaction rule. The stalled-action fraction falls to 25.9%, mean maximum depth reaches 3.0, and mean return rises from 26.62 to 883.77 on the same Episodes (Figure 6a). The resulting memory represents which interactions failed, allowing later planning to distinguish an obstructed route from an unexplored one.

The other Runs likewise repair navigation before concentrating on food, combat, and retreat. These rules can interfere: an attack that leaves the player in place must not be classified as a blocked movement. The later survival decisions remain a source of variation in final performance.

D.2 CRAFTER: PERCEPTION ERRORS PROPAGATE INTO PLANNING

All three Runs build visual representations of terrain, inventory, and crafting dependencies. Run 1's diagnostic code replays saved RGB observations through the policy to expose its inferred position and decision state. An image probe reveals that the world still looks asleep when the energy-based sleep flag has already cleared. Ignored movement commands then corrupt the policy's position estimate.

The S4–S5 revision adds image-based sleep detection and removes the premature energy-based reset, alongside changes to plant recognition, map confidence, and resource priorities. The composite achievement score rises from 35.41 to 61.32 on four matched Episodes. Figure 6b pairs the recorded sleeping appearance with the old and revised state-update rules.

Related errors recur in Run 2's object recognition under poor visibility and Run 3's sleep handling and oscillation between resource targets. Across the Runs, these failures connect perception errors to persistent planning state rather than to a single incorrect action.

D.3 BALATRO: INVESTING DISCARDS IN HAND UPGRADES

Astra uses Joker rules and game state to coordinate discards, hand selection, and purchases. In Run 1, inspection of recorded Joker rules precedes the S4–S5 revision, which adds deliberate use of Burnt Joker: the first discard of each round upgrades the discarded hand type. S5 targets the most-played hand type and chooses matching cards with low estimated value, preserving more valuable cards for play. In training Episode 5, the policy discards 3♢ at step 169, raising High Card from level 8 to 9. The next action plays High Card at level 9 (Figure 7a). This illustrates using a limited action resource to strengthen subsequent plays.

The revision also incorporates owned Joker effects into discard planning and adjusts purchase values to reflect hand-type usage. Scores rise from 18.875 to 23.750 on eight matched training Episodes and from 17.625 to 21.344 on 32 held-out Episodes, with no policy failures in either version. These are gains for the complete revision; the recorded upgrade illustrates the new strategy without isolating its contribution to the score increase.

Local gains do not always identify the strongest final candidate. In Run 1, S10 outperforms S9 on 24 matched training Episodes (23.67 versus 21.04), but their order reverses on 16 Validation Episodes (19.56 versus 23.81). The Host selects S9, which scores 20.38 in Assessment. Retaining alternatives helps here: the later revision is weaker on Validation, which occurs only after the agent exits.

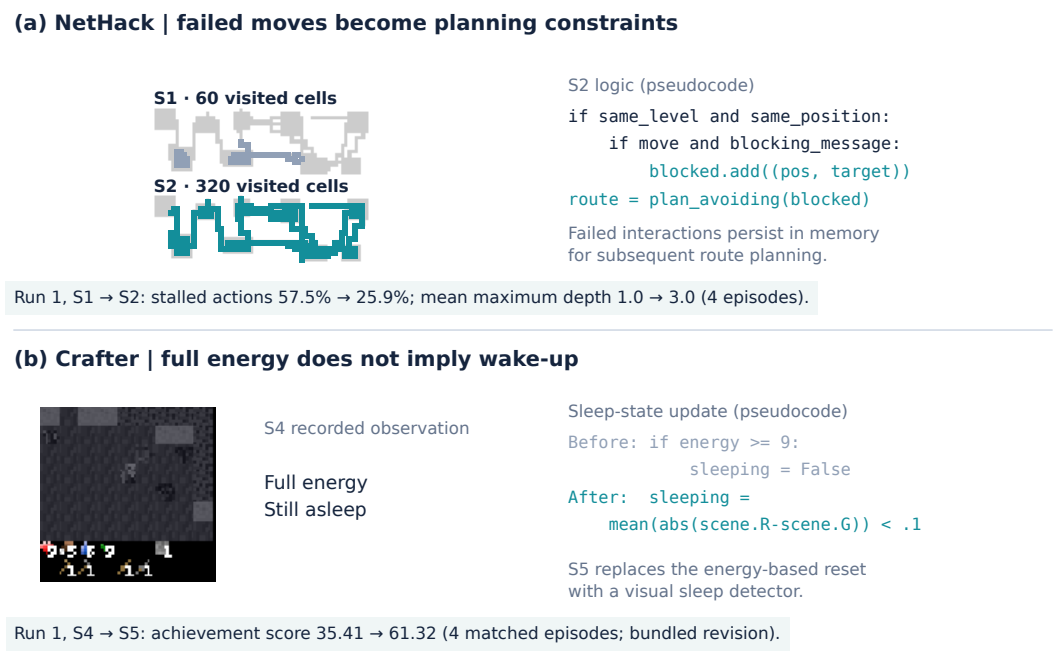


Figure 6: **From observed failure to policy-state updates.** (a) NetHack Run 1: S1/S2 maps show recorded visits in training Episode 1, on dungeon level 1. Gray cells were observed by either policy; colored cells were visited. Pseudocode summarizes S2's blocked-edge memory. (b) Crafter Run 1: an S4 observation from Episode 3 still shows sleep at full energy. The S5 code uses scene color to update the sleep state. Outcome strips summarize four matched training Episodes per comparison.

D.4 FrankaKitchen: the agent writes its own numerical search

All three Runs develop staged joint-waypoint controllers and then generate programs to search over their parameters. In Run 1, a generated program perturbs waypoints and retains candidates ranked by

$$U(\theta) = R(\theta) + 0.4 \sum_{k \notin C(\theta)} \max\left(0, 1 - \frac{d_k(\theta)}{d_k^{\text{ref}}}\right), \tag{6}$$

where R is Episode return, C is the set of completed tasks, and d_k is the minimum distance to subgoal k along the trajectory. Reference distances scale subgoals. The extra term ranks near misses with equal return; it changes the search objective, not the environment score.

A later coordinate search produces an unusually localized comparison. S61 and S71 differ only in the first joint target of the final waypoint, from -0.5792 to -0.6792. On the same Episode, the minimum kettle-goal distance falls from 0.3134 to 0.2997 and completed tasks increase from four to five (Figure 7b). The distance uses the full achieved and desired goal vectors. Here, detailed feedback supplies a useful signal inside a region where the task score is otherwise constant.

Across the three Runs, 291 of 308 submissions evaluate a single Episode. Many are iterations of generated search code. Final Assessment scores are 4.97, 3.00, and 4.00: local search improves parameters, but its outcome still depends on the motion sequence and subgoals chosen by the agent.

D.5 CarRacing: speed tuning must preserve completion

All three Runs build controllers around visual road geometry and speed estimation. Feedback helps separate perception errors from control errors. In Run 1, frame-displacement probes expose implausible speed estimates from repetitive ground texture, prompting continuity checks. Later development compares speed profiles, corner braking, and steering adjustments.

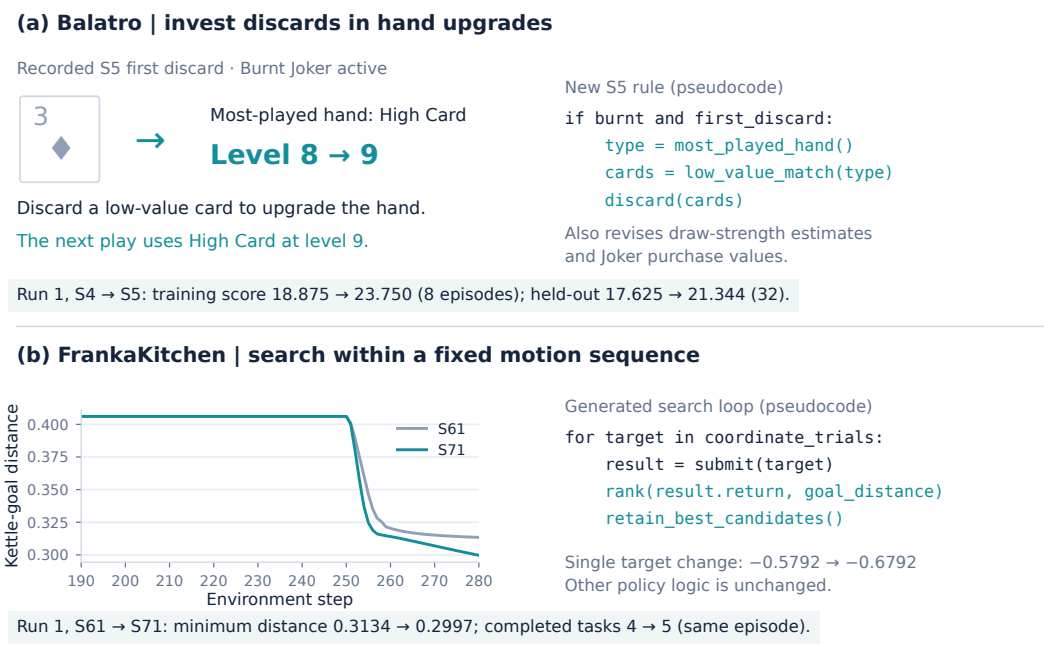


Figure 7: **Investing in hand upgrades and generating a search procedure.** (a) Balatro: a recorded S5 discard activates Burnt Joker, raising High Card from level 8 to 9 before the next play. Pseudocode summarizes the new discard rule; score changes cover the complete S4–S5 revision. (b) FrankaKitchen Run 1: the curves connect recorded kettle-goal distances on training Episode 0; S61/S71 differ in one joint target. Pseudocode summarizes the generated coordinate-search loop. Training and held-out results are identified separately in the outcome strips.

Once a controller can finish laps, mean return can conceal a loss of reliability. Run 2 evaluates S11 and S12 on the same 32 Episodes. Their mean returns are close—916.83 and 915.37—but completed laps fall from 32 to 29. Only four speed-control constants differ between the saved policies. Figure 8a shows small gains on most Episodes offset by three incomplete laps; all three reach the 1,000-step limit. S11 is ultimately selected. Completion counts therefore provide a useful second view of changes that appear nearly equivalent in aggregate return.

The other Runs likewise compare faster settings against their ability to finish unfamiliar tracks, sometimes restoring earlier choices. Final Assessment returns are 909.22, 917.37, and 927.67. At this level of performance, the remaining revisions largely refine an established controller, with small score differences that need to be read alongside failure outcomes.

D.6 TreantsForest: A long route is an imperfect planning objective

Astra constructs connected mazes and searches over layouts intended to delay the adventurer. The central difficulty is choosing a useful surrogate for that delay. A longer geometric route need not take longer to traverse when the layout reveals useful information early or offers few distracting branches. The first two Runs compare route-length criteria, layout pools, and online replanning; their logs repeatedly revisit the relationship between the planned maze and the adventurer's actual exploration.

Run 3 develops an objective that accounts for hidden, bent dead ends rather than relying only on distance. Its added term aggregates separation among unseen dead ends hidden behind bends, using a tree of shortest paths to the goal. Figure 8b shows the revised objective and recorded exploration paths on the first matched Episode. S11 and S13 score 1476.25 and 1600.17 on the same 12 training Episodes. Later trials continue adjusting the balance between these terms. The final Assessment scores are 1186.09, 1214.47, and 1555.66 across Runs 1–3. This case illustrates a revision to what the policy optimizes internally: feedback motivates a layout criterion closer to the behavior that determines the environment score.

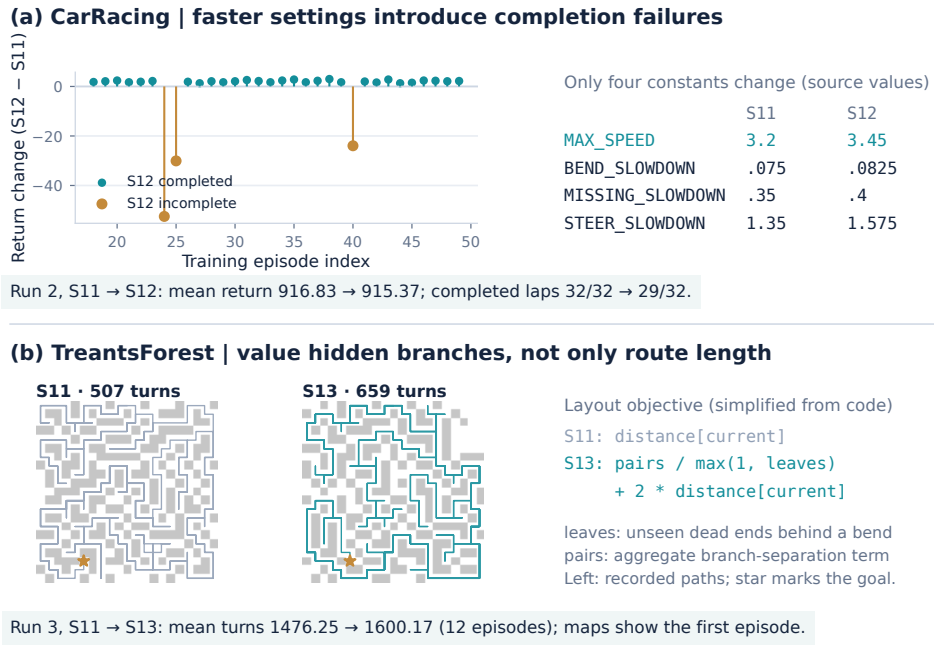


Figure 8: **How internal control and planning objectives affect behavior.** (a) CarRacing Run 2: each point is S12 minus S11 return on one of 32 matched training Episodes. Orange points mark incomplete S12 laps; the right panel lists all four changed constants. (b) TreantsForest Run 3: recorded paths and final wall layouts on training index 4, with the goal marked by a star. S13 adds a hidden-branch term to the layout objective. The displayed Episode lasts 507 versus 659 turns; the outcome strip reports all 12 matched Episodes.

D.7 TwoArmTransport: repairing transitions between working stages

Astra decomposes transport into lid removal, trash handling, grasping, handoff, and placement. End-effector and object traces help locate failures at transitions between these stages. A successful grasp is insufficient if the arms interfere during separation or the receiving arm loses the object before placement.

In Run 1, the S9–S10 revision moves the donor withdrawal target upward, removes a simultaneous receiving-arm lift, and advances release from step 480 to 465. Success increases from two to four of the same four Episodes, with mean return increasing from 5.50 to 25.75 (Figure 9a). Figure 5 shows the corresponding handoff frames. The key change concerns coordination between working stages: independent arm motions must also leave the receiving grasp stable during separation.

The other Runs expose different transition failures. In Run 2, success flags distinguish incomplete trash handling from successful payload placement. Run 3 revises receiving-arm reach and wrist motion after attempts to lift higher make transfer less reliable. Final Assessment scores are 102.03, 96.84, and 93.00. The development logs thus contain both sequence construction and repeated correction of interactions among its components.

D.8 Humanoid: diagnostic progress without sustained running

The humanoid Runs provide a useful contrast to the successful task-specific controllers. Astra probes action responses, repairs joint mappings, and searches over pose targets, periodic motions, and feedback gains. These activities produce executable controllers and some improvements in return, but the inspected logs do not show sustained running in any of the three Runs.

(a) TwoArmTransport | coordinate separation and release



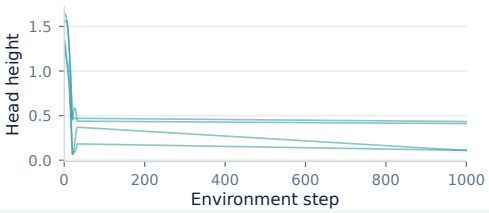
Three edits (source values; simplified names)

```
donor target at step 400:
  [0, −.25, 1.20] → [0, −.08, 1.40]
receiver lift +.15: removed
release step: 480 → 465
```

Keep the receiving arm in place while the donor withdraws upward.

Run 1, S9 → S10: successes 2/4 → 4/4; mean return 5.50 → 25.75 (matched episodes).

(b) Humanoid | higher return does not establish locomotion



S70 controller (pseudocode)

```
target = pose_with_balance_terms(up)
j = joint_mapping[i]
u = kp*(target[i]-q[j]) - kd*v[j]
u += orientation_feedback(up)
action[i] = clip(u, -1, 1)
```

Four recorded trajectories; eight episodes scored.

Run 1, zero action → S70: mean return 0.872 → 1.238 (8 episodes); no sustained running in logs.

Figure 9: **Coordination repairs and the limits of local controller search.** (a) TwoArmTransport Run 1: camera crops at step 416 of training Episode 0, paired with the three S9/S10 code edits. Success counts use four matched Episodes. (b) Humanoid Run 1: S70's control logic and head-height traces for the four traced training Episodes; lines connect recorded samples. The return comparison uses S67, a zero-action control, and S70 on the same Episodes.

Run 1 compares a zero-action control with three developed candidates on eight matched Episodes. Mean return rises from 0.872 for the control to 1.241, 1.199, and 1.238 for S68–S70. Figure 9b shows S70's control structure and head-height traces: the four traced Episodes lose their initial height early and remain low at the recorded later steps. The selected S70 achieves 1.178 in Assessment; the other two Runs finish at 1.112 and 1.075. The numerical gain is real on this comparison, but does not amount to the intended locomotion behavior. Unlike the waypoint structure in Kitchen or the staged controller in Transport, the representations explored here do not yield a reliable solution within the available interaction budget.