Evaluating a Terminal Task Is Itself a Terminal Task: What LLM Graders Need to Judge Agent Work

Tiancheng Ge[1,†] Jiakai Xu[1,†] Zhou Yu[1] Eugene Wu[1] Kostis Kaffes[1]
[1] Columbia University
tg2985@columbia.edu ax2155@columbia.edu zy2461@columbia.edu ewu@cs.columbia.edu
kkaffes@cs.columbia.edu
[†] Equal contribution.

Abstract

Benchmarks grade agents with verifiers written in advance, but when no verifier is available, an LLM critic must judge the agent's work. We ask what such a critic needs to grade terminal tasks, whose outcome depends on files, processes, and services that the agent's trajectory may not fully show. On all 89 tasks of Terminal-Bench 2.1, we vary when the critic judges, what environment access it has, and what reference it receives, and we compare its verdicts with the benchmark's verifier. All experiments run on Waypoint, a branchable execution environment. Judging unfinished work is hard: critics rank intermediate states only modestly better than chance, even with full access to the environment. Judging finished work goes better when the critic can run its own checks. A critic that may run any command, including tests, in a disposable fork of the final state, checks every requirement and gives a verdict on 92% of states, significantly more than critics that only read the trajectory or inspect the fork. A fork of the initial state also improves accuracy, because the critic can check requirements that refer to the original environment. Used as a final check before the agent submits, the critic raises pass@1 from 63.4% to 72.4%. In short, evaluating a terminal task is itself a terminal task.

1 Introduction

When an agent finishes a task, a grader is needed if we want to know whether it succeeded. On a benchmark, the verdict comes from a verifier that is written and verified for the task in advance. As more tasks that users or agents pose in practice have no such verifier, another LLM or agent is usually used as the grader, which we call the *critic*, to judge what the agent did. Tree search, reinforcement learning, and recursive self-improvement all rely on this signal to make decisions or to assign rewards, so the critic's accuracy matters.

LLMs today are no longer only chatbots, but agents that use tools and terminals to act on an environment. What a grader must judge changes accordingly, from the agent's trajectory alone to the state of the machine, including files, processes, installed packages, and databases about which no direct information may be found in the trajectory. Ideally, the grader is itself an agent that runs commands to check and test this state. Its commands, however, may change or even break the environment. The state that the agent handed in would then no longer be the state that the verifier grades if a critic agent worked on this state before, and their verdicts could not be compared. For example, GIT-MULTIBRANCH asks the agent to set up a Git server that deploys each pushed branch to a web server. The verifier checks this by pushing commits, and a grader can only perform the check in the same way, which modifies the server itself that the verifier will later use for grading. For this reason, agent-as-a-judge systems usually give the judge read-only tools (Zhuge et al., 2024). Branchable environments (Xu et al., 2025, 2026b;

Dong et al., 2026; Wu et al., 2026) remove this constraint: the critic can run any command in a disposable fork of the agent's state, while the verifier grades another fork of the same state.

This paper asks what a critic needs to grade such work. We compare critics with three levels of environment access, score each against the verifier on all 89 tasks of Terminal-Bench 2.1 (Marten et al., 2026), and ask three questions: How does the grading position (an intermediate state or a final state) change the behavior of critics, what is the impact of environment access, and what reference besides the final state does it need to collect enough evidence? We find:

- **Intermediate states carry little signal.** Critics rank them only modestly better than chance, and environment access helps little (§5.2). Tree search guided by such a critic does not significantly change pass@1 (§5).
- **Execution lets a critic give a verdict on final states.** Asked to check each requirement of the task with direct evidence, a critic that can run any commands comes up with a verdict on 92% of final states, against 43%–63% for critics that cannot. When they do give a verdict, the three critics are about equally accurate (80–84%; §5.2).
- **A fork of the initial state provides evidence that the final state lacks.** It lets the critic check requirements that refer to the original environment, which improves accuracy and makes the critic more consistent with itself (§5.3).
- **A critic as a final check is useful.** Used as a final check before the actor stops, the critic raises pass@1 from 63.4% to 72.4% (§6).

Together, these results support one observation: evaluating a terminal task is itself a terminal task. To grade the work, the critic has to run checks in the environment itself.

2 Related Work

Judging agent work. LLM judges do not always agree with the ground truth (Lù et al., 2025), and without execution they can follow stylistic cues (Jain et al., 2025). Judges that collect evidence from the environment do better (Gou et al., 2023; Zhuge et al., 2024; Shi et al., 2026), but a judge that acts on the state it grades must not change it. We give the judge a fork, so that it can run any command, and vary when it grades and what reference it has.

Graders inside agent systems. Tree search uses judgments of intermediate states (Zhou et al., 2024; Koh et al., 2024; Antoniades et al., 2024), and reward models rank completed work (Shum et al., 2025; Kwok et al., 2026). These systems are evaluated by their end-to-end gain (Yu et al., 2025; Stroebl et al., 2026); we instead score the grader itself at different stages of the trajectory, to explain why search fails on terminal tasks (§5).

Branchable environments. Fork and rollback for agent sandboxes were built for exploration and throughput (Xu et al., 2025, 2026b; Dong et al., 2026; Wu et al., 2026; Huang et al., 2026). We use them for evaluation (§3).

3 Branchable Environments for Evaluation

Careful evaluation of a terminal agent is a controlled experiment on the state of its execution environment, and a branchable environment provides the control. However, existing ways to save and restore the environment, such as container commits that capture only the filesystem and replay of the commands according to the agent's log, do not adequately support the evaluation of terminal agents (Xu et al., 2025). We argue that a branchable environment must meet four requirements at once to support controlled experiments.

Complete state. A snapshot must capture the whole environment, including running processes and their memory, not only its files. The outcome of a terminal task is not always written to files. On Terminal-Bench 2.1, of the 89 verifiers we annotated by hand, at least 13

of them inspect a live process or probe a service (Appendix A). For example, QEMU-ALPINE-SSH requires the virtual machine to keep running and to accept SSH connections. If a grader has to restart such a service itself, its judgment is no longer about the state that the agent handed in. Waypoint (Xu et al., 2026b), on which we ran every experiment, saves the process tree, its memory, and the terminal sessions with CRIU (CRIU Project, 2012) and the filesystem with OverlayFS, so every fork resumes with the agent's live processes still running.

Isolation. Live forks of an environment must not interfere with one another or with the snapshot itself. Graders may perform destructive operations, such as installing dependencies, running tests, or restarting services. If these operations ran on the graded environment, every later grader and the verifier would see a modified state that the actor did not produce. In Waypoint, each fork has its own isolated working environment, and changes in one fork do not affect the others. C^{act} can therefore run any command in its own fork, while the verifier grades an untouched copy (§5.1).

Efficient reuse. A snapshot must be reusable many times, and each use must be cheap and start from exactly the same state. In a careful evaluation we need to repeat the measurements many times from the same starting state. A command replay might also reproduce a state, but it re-executes every compilation, installation, and training run in the trajectory, so it takes about as long as the original run and may not reproduce the same state. A Waypoint snapshot is immutable and deterministic: a fork takes about 60 ms, determined by the size of the snapshot's memory rather than by how long the agent has run. Because forks are cheap, the three critics judge the same 406 final states and can be compared state by state (§5.2).

History. Earlier snapshots must stay available, and a critic must be able to use forks of several of them at once. During the evaluation, a grader may need an earlier state for comparison, such as a file that must stay unchanged or a program that must keep its original behavior. Waypoint keeps every snapshot in the session's history and can fork any of them at any time, running forks of several snapshots in parallel. $C^{\text{act}}{+}S_0$ therefore receives forks of both the initial state S_0 and the final state and compares the two directly (§5.3).

4 Setup

We use all 89 tasks of Terminal-Bench 2.1 in every experiment. We separate three roles: the *actor*, the *critic*, and the *search policy*. The actor is the Terminus-2 agent provided by Harbor (Harbor Framework Team, 2026), which completes the tasks. A critic estimates the value of a state from the task instruction, the actor's trajectory up to that state, and its own level of access to the environment of that state (§5.1; prompts in Appendix B). The search policy decides when to stop, when to branch, when to call a critic, and from which state the actor continues. The ground-truth verdict on a task is given by the benchmark's own verifier, and neither the actor nor the critic sees the verifier or its output.

We run each task five times and report pass@1 with 95% bootstrap intervals over tasks. Each run is limited by the task's own time limit T, which includes all the extra cost of search, such as critic calls and forking. Every LLM call goes to GPT-5.6 Luna on Amazon Bedrock. We compute API cost from the tokens consumed, at Bedrock's list prices as of September 24, 2026: $0.25, $0.02, and $1.20 per million uncached input, cached input, and output tokens. The code for the experiments and the prompts of all agents are at https://github.com/Alex-XJK/harbor-StateFork/tree/arxiv-v1.

5 What a Grader Needs: Position, Access, and Reference

Our first attempt gives the branching tool to the actor directly, to see whether it can use the tool to complete more tasks. Across all 445 trials, however, the actor calls the tool only 3 times, and pass@1 is 61.0%, against 63.4% without the branching tool. The actor does not use it actively and effectively when given the tool alone. We next let MCTS (Kocsis and Szepesvári,

2006), guided by an LLM critic, decide when to branch and which state to continue. MCTS branches in 99.3% of trials, but pass@1 is only 64.0% (difference $[-4.5, +5.5]$), at 3.81$\times$ the baseline's cost ($40.79 against $10.70). To explain why search fails, we measure its critic directly.

5.1 Critics and How We Score Them

Critic variants. The three critics differ in their access to the environment. C^{traj} is a plain LLM-as-a-judge: it sees the actor's trajectory and has no tools. C^{read} is able to use read-only tools in a fork of the state, such as `cat`, `ls`, `grep`, and `diff`, which is the setting of most agent-as-a-judge systems (Zhuge et al., 2024). These systems keep the judge read-only because its writes would change the state that the verifier grades later. With branching, we instead get two identical copies of the state, one for the critic and one for the verifier. This gives a true agent-as-a-judge, C^{act}, which may run any command in its fork, including installing packages, running tests, and destructive actions, just as a verifier may do.

Positions and metrics. Search uses a critic to compare intermediate states, so we evaluate these three critics at 25%, 50%, and 75% of the actor's trajectory. At such a state S_t, the critic estimates the chance $\hat{q}(S_t)$ that the actor, continuing from S_t, will eventually succeed. We run each task five times independently and ask critics to grade at these three positions of each run. We estimate each ground truth $q(S_t)$ as the pass rate of 5 continuations from S_t: the run's own continuation and 4 additional independent rollouts with the same actor and remaining budget. Because a critic in search is responsible for picking the better branch, our main metric is within-task pairwise accuracy. We also evaluate the critics at the final state that the actor submits. For the final state, the critic's role changes from prediction to verification: it lists every requirement of the task first, marks each as met, unmet, or unable to check, with its reasons, and may pass the state only if every requirement is met. We score this verdict $\hat{v}$ against the task's own verifier. In total, we collect valid data from 1111 intermediate and 406 final states.

5.2 When to Grade: Intermediate States Carry Little Signal

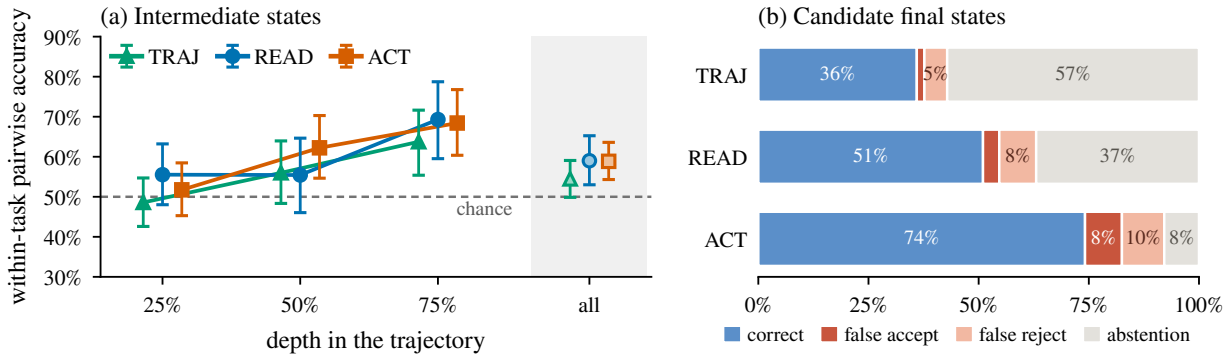


Figure 1 | Critics by position in the actor's trajectory, on 89 Terminal-Bench 2.1 tasks. TRAJ, READ, and ACT are C^{traj}, C^{read}, and C^{act}. **(a)** Within-task pairwise accuracy: among pairs of same-task, same-depth states, the share in which a critic gives the higher $\hat{q}$ to the state whose 5 continuations pass more often; *all* pools the depths, and the dashed line marks chance (50%). Intervals are 95% bootstrap over tasks. **(b)** Every verdict $\hat{v}$ on the 406 states, against the verifier. A false accept fails to find a violation on a failing state; a false reject names a violation on a passing state; an abstention gives no verdict because at least one requirement could not be checked.

As Figure 1a shows, no critic ranks intermediate states meaningfully better than chance at 25% or 50% of the trajectory, even though the same model, as the actor, passes 63.4% of the tasks. This is a serious problem for search: a branch with a better strategy is hard to identify at an early stage where choosing it matters most. At 75% depth, the critics rank states better than

chance, but the signal is still limited (64%–69%). Environment access makes little difference: pooled over the depths, C^{act} beats C^{traj} by only $+4.4$ points $[-0.9, +9.9]$.

On the contrary, environment access does matter at the final state, where running checks is often necessary (Figure 1b). Without execution, critics often abstain because they cannot check some requirement: C^{traj} and C^{read} abstain on 57% and 37% of states, against 8% for C^{act}. Execution thus lets C^{act} give a verdict on 92% of states, against 63% for C^{read} and 43% for C^{traj}. When they do give a verdict, the three critics are about equally accurate: their verdicts match the verifier on 84%, 81%, and 80% of these states. Environment access therefore expands the set of states that a critic can judge rather than making each judgment more accurate. Appendix D gives the full analysis, including an audit of the violations that each critic names and a breakdown by verifier type.

Among intermediate states, critics give better estimation at a later position in the trajectory. The deeper the state, the more of the actor's work the critic can see and the less work remains to be predicted. More fundamentally, judging an intermediate state is a prediction, and there is no general principle to guide it. A critic can roughly tell that the actor has completed part of the task or is heading in the right direction, but it can be hard to predict how likely the actor succeeds at the end without mistake. In contrast, the principle for judging a final state is clear: check whether the state satisfies each requirement of the task. The state is static as the actor is not allowed to do any more changes; there is no need to predict what the actor may do in the future. Unless the task is vague, each requirement can map to a check, so the critic can confirm all of them or point to the one that fails the task.

5.3 What Reference the Grader Has: The Initial State

When we annotated the critics' trajectories to find the causes of false verdicts and abstentions, many failures involved requirements that need to check the difference between the initial and the final state: a file must stay unchanged, a change must follow certain rules, or a program must keep its original behavior. With access only to the final state, a critic can hardly check such requirements, because the task instruction and the final state alone do not show all the changes. The actor's trajectory in the critic's prompt may describe some of the changes, but critics rely more on direct evidence from the state than on the actor's own account. On the 406 final states of Figure 1b, such requirements account for 15 of the 56 verdicts in which C^{act} does not accept a passing state. We therefore introduce $C^{\text{act}}{+}S_0$, which also receives a live fork of the initial state, and compare it with C^{act}. We rebuild the final states of all 89 tasks and let each critic judge every state twice, which also measures how consistent each critic is with itself.

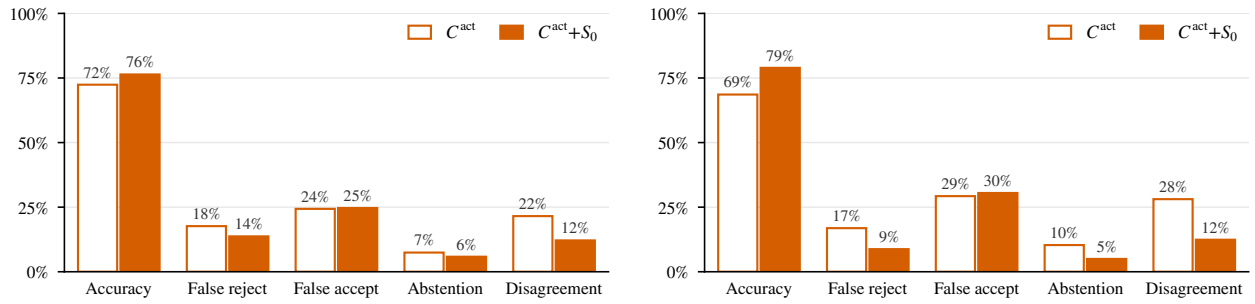


Figure 2 | C^{act} against $C^{\text{act}}{+}S_0$. **Left:** all 404 candidate final states. **Right:** the 121 states on which $C^{\text{act}}{+}S_0$ used the fork of S_0 in at least one of its two evaluations. Every rate counts both verdicts per state. Accuracy is the share of all verdicts that are correct; an abstention (as in Figure 1b) counts as incorrect. False reject is the share of verdicts on passing states that name a violation, and false accept the share of verdicts on failing states that accept. Abstention is the share of all verdicts. Disagreement is the share of states on which a critic's two verdicts differ in kind (accept, named violation, or abstention).

Over all 404 states that we rebuilt successfully (Figure 2, left), $C^{\text{act}}{+}S_0$ improves accuracy by $+4.0$ points [$+0.7$, $+7.3$]. The gain comes mainly from fewer false rejects, which fall from 18% to 14% of the verdicts on passing states (-4.0 points [-7.3, -0.8]), while false accepts do not rise ($+0.3$ points [-4.4, $+5.0$]). Abstentions change little (-1.6 points [-4.5, $+1.0$]), as C^{act} already abstains on only 7% of its verdicts. The critic also disagrees with itself less often, on 12% of states instead of 22%. When we concentrate on the 121 states in 25% of all evaluations, where $C^{\text{act}}{+}S_0$ used the initial fork, accuracy rises by $+10.3$ points [$+2.3$, $+18.9$] there, against $+1.2$ points [-1.4, $+3.9$] on the other states that $C^{\text{act}}{+}S_0$ did not really use the initial fork (Figure 2, right). False reject, abstention and disagreement also drop against $C^{\text{act}}{+}S_0$ in all 404 states. Through the trajectory analysis of $C^{\text{act}}{+}S_0$, we can also find that giving access to the initial state helps agents check the requirements that they lacked methods to check directly before. In MODERNIZE-SCIENTIFIC-STACK, for example, C^{act} cannot tell whether the legacy file was preserved and abstains on that state, while $C^{\text{act}}{+}S_0$ compares the file's hash in both states and correctly accepts it.

6 The Critic as a Final Check

A critic that runs checks in a fork is useful when it judges a final state (§5.2) and can compare it with the initial state (§5.3). We therefore design Probe-and-Resume, which places $C^{\text{act}}{+}S_0$ in the actor's loop as a final check. Each time the actor claims to be done, a critic checks its work in disposable forks of that state and of S_0. If the critic rejects, the actor receives the reasons and resumes from the state it claimed (Xu et al., 2026a).

	Pass@1	Δ vs. Probe-and-Resume	Total model cost
Baseline	63.4%	-9.0	\$10.70
Probe-and-Resume	**72.4%**	–	\$23.14
Restart from S_0	66.3%	-6.1	\$38.12
Critic without S_0	64.5%	-7.9	\$34.77
Actor as critic	65.2%	-7.2	\$24.73
Codex CLI (reference)	76.6%	–	\$50.29

Table 1 | Pass@1 on Terminal-Bench 2.1 (five trials per task) and model cost of one trial of every task. *Baseline*: the actor's first claimed state. *Restart from* S_0: after each rejection, the actor restarts from S_0 instead of resuming from the state it claimed. *Critic without* S_0: the critic gets no fork of the initial state. *Actor as critic*: the actor judges its own claims within its own conversation instead of a fresh critic. Codex CLI: the same model in the Codex CLI, from Harbor Hub's public leaderboard (different harness and cost accounting).

As shown in Table 1, on all 89 tasks of Terminal-Bench 2.1, Probe-and-Resume raises pass@1 from 63.4% to 72.4% ($+9.0$ points [$+5.6$, $+12.6$]) within each task's time limit, which also covers every critic call. This is only 4.2 points below Codex CLI (76.6%), a more advanced agent that runs the same model, and costs less than half of it. Of the 136 first claims that fail, the critic correctly rejects 103 (76%), and the actor later fixes 43 of them (42%). Every ablation in Table 1 lowers pass@1 (Appendix E), and the gain also holds with Claude Sonnet 5 or GLM-5 as the critic and on SWE-bench Pro (Deng et al., 2025) (Appendix F).

7 Conclusion

We asked what an LLM critic needs to grade terminal tasks. Judging intermediate states is hard: critics rank them only modestly better than chance. At a final state, a critic that can run any command in a fork of the final state checks every requirement and gives a verdict on 92% of states, significantly more than critics that cannot. A fork of the initial state further improves the accuracy and makes the critic more consistent with itself. Used as a final check, such a critic raises pass@1 from 63.4% to 72.4%.

Acknowledgments

This research received DAPLab corporate support in the form of funding and/or compute from Amazon, IntellectAI, Infosys, Tidalwave, Veris, Shopify, Microsoft, Thinking Machines, Dandy, Perplexity, and Daytona. The views and conclusions presented here are those of the authors and should not be interpreted as representing the official positions of the funding organizations.

References

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. SWE-Search: Enhancing software agents with Monte Carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*. https://arxiv.org/abs/2410.20285. Cited at: 1.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525. https://doi.org/10.1109/TSE.2014.2372785. Cited at: 1.

CRIU Project. Checkpoint/Restore In Userspace (CRIU). https://criu.org/, 2012. Accessed: 2026-10-24. Cited at: 1.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks?, 2025. https://arxiv.org/abs/2509.16941. Cited at: 1, 2, 3.

Yunpeng Dong, Jingkai He, Shiqi Liu, Yuze Hou, Dong Du, Zhonghu Xu, et al. 2026. DeltaBox: Scaling stateful AI agents with millisecond-level sandbox checkpoint/rollback. *arXiv preprint arXiv:2605.22781*. https://arxiv.org/abs/2605.22781. Cited at: 1, 2.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, et al. CRITIC: Large language models can self-correct with tool-interactive critiquing, 2023. https://arxiv.org/abs/2305.11738. Cited at: 1.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. https://doi.org/10.5281/zenodo.20953922. Cited at: 1.

Yutong Huang, Vikranth Srivatsa, Alex Asch, Hansin Tushar Patwa, and Yiying Zhang. 2026. TClone: Low-latency forking of live GUI environments for computer-use agents. *arXiv preprint arXiv:2605.17320*. https://arxiv.org/abs/2605.17320. Cited at: 1, 2.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. In *Conference on Language Modeling (COLM)*, 2025. https://arxiv.org/abs/2504.07164. Cited at: 1.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, page 282–293, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 354045375X. https://doi.org/10.1007/11871842_29. Cited at: 1.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. 2024. Tree search for language model agents. *arXiv preprint arXiv:2407.01476*. https://arxiv.org/abs/2407.01476. Cited at: 1.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, et al. LLM-as-a-Verifier: A general-purpose verification framework, 2026. https://arxiv.org/abs/2607.05391. Cited at: 1.

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, et al. AgentRewardBench: Evaluating automatic evaluations of web agent trajectories. In *Second Conference on Language Modeling*, 2025. https://openreview.net/forum?id=fQcUZMPIvu. Cited at: 1.

Ryan Marten, Alex Shaw, Ivan Bercovich, Benedikt Droste, Tommaso Cerruti, Steven Dillmann, et al. Terminal-Bench, August 2026. https://doi.org/10.5281/zenodo.22105680. Cited at: 1.

Wentao Shi, Yu Wang, Yuyang Zhao, Yuxin Chen, Fuli Feng, Xueyuan Hao, et al. AJ-bench: Benchmarking agent-as-a-judge for environment-aware evaluation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Findings of the Association for Computational Linguistics: ACL 2026*, pages 25371–25413, San Diego, California, United States, July 2026.

Association for Computational Linguistics. ISBN 979-8-89176-395-1. https://doi.org/10.18653/v1/2026.findings-acl.1269. Cited at: 1.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, et al. 2025. SWE-RM: Execution-free feedback for software engineering agents. *arXiv preprint arXiv:2512.21919*. https://arxiv.org/abs/2512.21919. Cited at: 1.

Benedikt Stroebl, Sayash Kapoor, and Arvind Narayanan. The limits of inference scaling through resampling. In *The Fourteenth International Conference on Learning Representations*, 2026. https://openreview.net/forum?id=j8H84v6AZ1. Cited at: 1.

Tianyuan Wu, Chaokun Chang, Lunxi Cao, Wei Gao, and Wei Wang. 2026. Crab: A semantics-aware checkpoint/restore runtime for agent sandboxes. *arXiv preprint arXiv:2604.28138*. https://arxiv.org/abs/2604.28138. Cited at: 1, 2.

Jiakai Xu, Tianle Zhou, Eugene Wu, and Kostis Kaffes. 2025. Toward systems foundations for agentic exploration. *arXiv preprint arXiv:2510.05556*. https://arxiv.org/abs/2510.05556. Presented at the 1st Workshop on Systems for Agentic AI (SAA '25), co-located with SOSP 2025. Cited at: 1, 2, 3.

Jiakai Xu, Tiancheng Ge, Tujie Guan, Yusheng Li, Rohan Timmaraju, Eugene Wu, et al. Probe-and-Resume: When and how to evaluate agent execution, 2026a. https://arxiv.org/abs/2610.21088. Cited at: 1, 2, 3, 4.

Jiakai Xu, Tianle Zhou, Georgios Liargkovas, Danielle Gillai, Ruizhe Fu, Patrick Shen, et al. 2026b. StateFork: Branchable infrastructure for agent exploration. *arXiv preprint arXiv:2609.38648*. https://arxiv.org/abs/2609.38648. Cited at: 1, 2, 3.

Fei Yu, Yingru Li, and Benyou Wang. Scaling flaws of verifier-guided search in mathematical reasoning, 2025. https://arxiv.org/abs/2502.00271. Cited at: 1.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *PMLR*, 2024. https://proceedings.mlr.press/v235/zhou24r.html. Cited at: 1.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, et al. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*. https://arxiv.org/abs/2410.10934. Cited at: 1, 2, 3.

A Task List, Verifier Annotation and Timeouts

Table A1 lists the 89 tasks of Terminal-Bench 2.1, the time limit T each task gives the actor, and what each task's verifier does to grade it.

We read every verifier by hand. For each assertion it makes, we asked what a checker must do, at minimum, to settle it: read something that already exists in the environment (R), run something to produce a new observation (X), or compare against a value that only the verifier holds (O). R and X have finer codes. R1: file content, such as a file that exists or whose content or hash matches. R2: filesystem metadata, such as permissions, owners, timestamps or git objects. R3: runtime state, such as a live process, a listening port or an environment variable. X1: run a binary, a test or a computation and check its output. X2: probe a service, such as connecting to a port or sending a request. A task carries every code its assertions need.

We group tasks by their strongest code: *holds the answer* if any assertion is O (19 tasks), otherwise *runs* if any is X (48), otherwise *reads* (22); Appendix D reports critic agreement by group. The last column marks the 47 tasks whose instruction states a requirement relative to the initial state S_0, such as a file that must stay unchanged or an output that must match the original program's.

Task	T (min)	Verifier	S_0
`adaptive-rejection-sampler`	15	R1, X1	
`bn-fit-modify`	60	R1, O	
`break-filter-js-from-html`	20	R1, X1	
`build-cython-ext`	15	R1, X1, O	✓
`build-pmars`	15	R1, X1	
`build-pov-ray`	200	R1, X1, O	✓
`caffe-cifar-10`	60	R1, R2, X1	
`cancel-async-tasks`	15	R1, X1	
`chess-best-move`	15	R1	✓
`circuit-fibsqrt`	60	R1, X1	✓
`cobol-modernization`	15	R1, X1	✓
`code-from-image`	20	R1	✓
`compile-compcert`	40	R1, R2, X1	
`configure-git-webserver`	15	X1, X2	
`constraints-scheduling`	20	R1	✓
`count-dataset-tokens`	15	R1, O	
`crack-7z-hash`	30	R1	✓
`custom-memory-heap-crash`	30	R1, X1	✓
`db-wal-recovery`	15	R1	✓
`distribution-search`	60	R1	
`dna-assembly`	30	R1, X1	✓
`dna-insert`	30	R1, X1	✓
`extract-elf`	15	R1, X1, O	
`extract-moves-from-video`	30	R1	✓
`feal-differential-cryptanalysis`	30	X1, O	
`feal-linear-cryptanalysis`	30	R1	✓
`filter-js-from-html`	30	R1, X1, X2, O	

Table A1 | Continued

Task	T (min)	Verifier	S_0
financial-document-processor	20	R1	✓
fix-code-vulnerability	15	R1, X1	✓
fix-git	15	R1	✓
fix-ocaml-gc	60	R1, X1, X2, O	
gcode-to-text	15	R1	✓
git-leak-recovery	15	R1, R2	✓
git-multibranch	15	X2	
gpt2-codegolf	15	R2, X1	✓
headless-terminal	15	R1, X1, X2	
hf-model-inference	15	R1, X1, X2	
install-windows-3.11	60	R1, R3, X2	✓
kv-store-grpc	15	R1, R3, X2	
large-scale-text-editing	20	R1, X1	✓
largest-eigenval	15	X1	
llm-inference-batching-scheduler	30	R1	✓
log-summary-date-ranges	15	R1	✓
mailman	30	R1, X2	
make-doom-for-mips	15	R1, X1	✓
make-mips-interpreter	30	R1, X1	✓
mcmc-sampling-stan	30	R1, X1	✓
merge-diff-arc-agi-task	15	R1, X1, O	✓
model-extraction-relu-logits	15	R1, X1	
modernize-scientific-stack	10	X1	✓
mteb-leaderboard	60	R1, O	
mteb-retrieve	30	R1	✓
multi-source-data-merger	15	R1	✓
nginx-request-logging	15	R1, X1, X2	
openssl-selfsigned-cert	15	R1, R2, X1	
overfull-hbox	12.5	R1, X1	✓
password-recovery	15	R1	✓
path-tracing	30	R1, X1	✓
path-tracing-reverse	30	R1, X1	✓
polyglot-c-py	15	X1	
polyglot-rust-c	15	R1, X1	
portfolio-optimization	60	R1, X1	✓
protein-assembly	30	R1, O	✓
prove-plus-comm	15	R1, X1	
pypi-server	15	R1, X1, X2	
pytorch-model-cli	15	R1, X1, O	
pytorch-model-recovery	15	R1, X1	✓
qemu-alpine-ssh	15	R3, X2	

Table A1 | Continued

Task	T (min)	Verifier	S_0
`qemu-startup`	15	R3, X2	
`query-optimize`	15	R1, X1, O	✓
`raman-fitting`	15	R1	✓
`regex-chess`	60	R1, X1	
`regex-log`	15	R1, X1, O	
`reshard-c4-data`	60	R1, R2, X1, O	
`rstan-to-pystan`	30	R1	✓
`sam-cell-seg`	120	R1, X1, O	
`sanitize-git-repo`	15	R1, R2	✓
`schemelike-metacircular-eval`	40	R1, X1	
`sparql-university`	15	R1, X1, O	
`sqlite-db-truncate`	15	R1	✓
`sqlite-with-gcov`	15	R1, R2, X1	
`torch-pipeline-parallelism`	15	R1, X1	
`torch-tensor-parallelism`	15	R1, X1	
`train-fasttext`	60	R1, X1, O	
`tune-mjcf`	15	R1, X1	✓
`video-processing`	60	R1, X1, O	
`vulnerable-secret`	15	R1	✓
`winning-avg-corewars`	60	R1, X1	✓
`write-compressor`	15	R1, X1	✓

Table A1 | The 89 tasks of Terminal-Bench 2.1. T: the task's time limit for the actor. *Verifier*: every code its assertions need (R1–R3 read, X1–X2 run, O compares with a value only the verifier holds); tasks are grouped by the strongest one. S_0: the task states a requirement relative to the initial state (§5.3).

B Prompts

All prompts used in this paper are in the repository at https://github.com/Alex-XJK/harbor-StateFork/tree/arxiv-v1. The actor's prompts are under `src/harbor/search/templates/`, whereas the critics' prompts and the feedback template are under `src/harbor/search/critics/templates/`. The code in the same directories shows how each prompt is built from its template, the task and the trajectory.

What each prompt asks. The actor runs under the standard Terminus-2 prompt and is not told about the critic; it claims completion by calling `task_complete`. Every critic receives the task instruction and the actor's trajectory $\tau_{\leq t}$. At an intermediate state, it returns an estimate $\hat{q}(S_t)$ of continuation success. At a candidate final state, it lists the task's requirements, settles each with a check it can cite, marks it met, unmet or not established, and may accept only if every requirement is met (Appendix D.1). The critics differ in their tools: C^{traj} has none, C^{read} runs read-only commands in a fork of the state, C^{act} runs arbitrary commands in it, and $C^{\text{act}}+S_0$ also gets a fork of the initial state. All three adapt one verifier template to the critic's tools and share its rules for reading the task and its report format. On a rejection during the

final check of §6, the actor receives the failed requirements, why they failed and how the critic checked, and is asked to address them or to run a command showing that a failed requirement holds.

C Statistical Methodology

Every interval in the paper is a 95% percentile bootstrap over tasks. Tasks are resampled with replacement 5000 times with a fixed seed; all trials, states and probes of a drawn task move together, and the statistic is recomputed on each resample. Differences between two configurations are paired before resampling: by trial when both configurations are read off the same trials (the baseline and the submitted state of Table 1), and by state when two critics judge the same state (§5.1, §5.3). Sample sizes are given with each table and figure.

D Committed Verdicts at Final States

D.1 Committed Verdicts

Every critic reports each requirement of the task as met, unmet or not established, and its prompt allows an accept only if every requirement is met (Appendix B). We classify each verdict from this report: an *accept*; a *named violation*, a reject that marks at least one requirement unmet; or an *abstention*, a reject that marks none unmet and at least one not established. Accepts and named violations are *committed*. Of the 3×406 verdicts, 4 reject although every requirement is marked met; they count as neither committed verdicts nor abstentions. No critic accepted with a requirement marked unmet or not established. The classification uses the critics' recorded reports and makes no new model calls. We do not ask for a chance of success at final states: it would answer a different question, how often states like this one pass, and it would give the actor nothing to act on.

Table D1 gives coverage, agreement and the pass rate of abstentions with their 95% bootstrap intervals over tasks (Appendix C). Paired gaps over the same states: coverage, C^{act} minus C^{read} $+29.1$ points $[+22.5, +35.9]$ and minus C^{traj} $+49.3$ points $[+42.8, +56.0]$; agreement, -0.4 points $[-6.1, +5.0]$ and -3.4 points $[-10.0, +3.4]$.

Abstentions carry no information. The states a critic abstains on pass at 52–65%, close to the base rate of 62%: on them, the critic knows nothing that a prior would not tell it.

Shared states. Each critic commits on different states. On the 122 states where all three commit, the paired gaps in agreement are -4.1 points $[-9.4, +0.6]$ for C^{act} minus C^{traj} and -1.6 points $[-7.0, +3.4]$ for C^{act} minus C^{read}.

Verifier groups. Table D2 gives agreement by verifier group. The gap between *holds the answer* and the other two groups is -19.7 points $[-52.3, +12.8]$ for C^{traj}, -17.0 points $[-38.8, +2.2]$ for C^{read} and -17.8 points $[-37.0, -1.1]$ for C^{act}. Agreement thus varies more with the verifier group than with the critic: committed verdicts agree with the verifier on 88–90% of states when it reads files, 81–86% when it runs the artifact or probes a service, and 66–68% when it holds a reference value that cannot be derived from the environment. This evidence is weak: the last group has only 19 tasks, and its gap to the other groups excludes zero only for C^{act}. If the gap holds, part of what pass or fail measures on these tasks is agreement with a value that only the verifier has. A grader can check such a value only by solving the task again, a case that software testing calls the oracle problem (Barr et al., 2015). Grader error would then be better reported by verifier type, since an aggregate mixes errors that a better grader could remove with errors that it could not.

Stability. Each critic judged each of these states once. For C^{act}, the reference-state experiment (Xu et al., 2026a, App. I) gives two further verdicts on each of 407 replayed states. Coverage is 92.9% and 92.4% and agreement 77.8% and 79.0% in the two passes, against 92% and 80% above. The two verdicts on a state are of the same kind (accept, named violation or abstention) on 78% of states, and agree on whether the verdict commits on 91%. The ag-

gregates are therefore stable, while single verdicts are not. We have no repeated verdicts for C^{traj} or C^{read}.

Within-task pairs. A critic could also approach the base rate by judging how hard a task looks rather than the state in front of it, because pass rates differ mostly between tasks: on 52 of the 86 tasks with more than one trial, all trials share one outcome. Such a critic cannot separate two states of the same task. The 34 tasks whose trials both pass and fail hold 160 states and 144 pairs of a passing and a failing state of the same task. A critic tells a pair apart if it accepts one state and names a violation on the other. Among those pairs, the share in which it accepts the passing state is 100% [100%, 100%] for C^{traj} (24 of 24), 95% [84%, 100%] for C^{read} (37 of 39), and 96% [90%, 100%] for C^{act} (79 of 82), with 95% bootstrap intervals over these tasks. A verdict that depends only on the task tells no pair apart. Every critic accepts the passing state in most pairs it tells apart, but C^{traj} and C^{read} tell fewer pairs apart, because they more often abstain.

	C^{traj}	C^{read}	C^{act}
Coverage (committed verdicts / states)	43% [36%, 50%]	63% [56%, 70%]	92% [88%, 96%]
Agreement (matching verifier / committed)	84% [74%, 92%]	81% [74%, 88%]	80% [74%, 86%]
Agreement on the 122 shared states	84% [74%, 93%]	82% [71%, 91%]	80% [69%, 90%]
Pass rate of abstentions	55% [44%, 66%]	65% [52%, 77%]	52% [30%, 78%]

Table D1 | Committed verdicts on the 406 candidate final states of Figure 1b, with 95% bootstrap intervals over tasks. Shared states: those on which all three critics commit. The base rate is 62% passing.

Verifier group	C^{traj}	C^{read}	C^{act}
reads (22 tasks)	90% [76%, 98%]	88% [80%, 95%]	90% [81%, 97%]
runs (48 tasks)	86% [75%, 95%]	82% [72%, 92%]	81% [73%, 88%]
holds the answer (19 tasks)	68% [36%, 100%]	67% [47%, 85%]	66% [48%, 82%]

Table D2 | Agreement of committed verdicts with the verifier by verifier group, with 95% bootstrap intervals over tasks.

D.2 Audit of Named Violations

A reject commits when the critic marks a requirement unmet (Appendix D.1). The audit asks whether each such mark rests on an observed violation or only relabels missing evidence. It makes no new critic calls.

What is audited. All 692 requirements marked unmet in the 330 rejects of the three critics: 101 of C^{traj}, 287 of C^{read}, and 304 of C^{act}.

What the judge sees. For each item, the judge gets the requirement as the critic states it and the critic's check, that is, what the critic did and what it found. As context, it also gets the critic's overall reasoning and its other requirements with their status. Verdicts carry opaque identifiers and are shuffled across critics, so the judge does not know which critic wrote an item. It does not see the verifier's result, the verifier's code, or the environment. It is told not to decide whether the task was solved and not to solve it again, only to judge whether the check is evidence of a violation.

Labels. Each item gets exactly one label.

- *Observed*: the check reports a concrete observation about the submitted work that contradicts the requirement, and the critic treats it as settling the matter. Examples are a value outside the required range; a missing file or wrong content; a crash, an error, or wrong

output on an input; a count or format that does not match; a forbidden artifact; or a fact about the artifact that conflicts with the literal wording. For instance, a file required to be a binary executable starts with a Python shebang.

- *Potential*: the check argues that the requirement could fail without observing a failure. Examples are a code path that might produce a violating output on some input (a fallback branch, a broad exception handler), an untested edge case, or a risk on hidden inputs. For instance, "the no-contour branch returns a rectangle".
- *Unproven*: the check only reports that the requirement could not be confirmed. This covers missing or insufficient evidence; reliance on the agent's claim; a failed tool or script; and an inconclusive independent attempt, including one whose contrary result the critic itself calls too unreliable to name a violation. It also covers an observation about the source material or about the critic's ability to verify, rather than about the submitted output. For instance, "comparison of sampled frames did not positively establish every move".

Three rules break ties. A check that mixes an observed violation with unproven parts is *observed*. Between *observed* and *potential*, the question is whether the failure was seen on the current state or only predicted. Between *potential* and *unproven*, the question is whether the critic names a specific way the requirement fails or only a lack of evidence.

Judges. Two judges label every item independently: Claude Opus 5.5, which we report, and Claude Sonnet 5.5. Both come from a different model family than the critics. Each judge reads the items in batches of about 80 verdicts and labels each one itself; labels may not be assigned by keyword rules. The judges agree on 94% of items (Cohen's $\kappa = 0.71$). Most of their disagreements (28) are between *observed* and *potential*. The labels, the rubric, and the script that produces the numbers below are in the repository at https://github.com/Alex-XJK/harbor-StateFork/tree/arxiv-v1.

Re-count. The paper's rule counts a reject as committed if it marks any requirement unmet. The *lenient* rule requires at least one unmet requirement labelled *observed* or *potential*, and the *strict* rule at least one labelled *observed*. A reject that fails the rule becomes an abstention. Table D3 gives coverage and agreement under each rule. Under the strict rule, the paired gaps for C^{act} minus C^{read} are $+35.7$ points $[+28.6, +43.1]$ in coverage and -2.4 points $[-8.6, +3.6]$ in agreement, and for C^{act} minus C^{traj} they are $+51.2$ points $[+44.2, +58.4]$ and -5.3 points $[-12.0, +1.6]$. On the 106 states where all three critics commit under the strict rule, agreement is 85–87%.

		Judge A (Opus 5.5)			Judge B (Sonnet 5.5)
		paper's rule	lenient	strict	strict
C^{traj}	Coverage	43% [36%, 50%]	42%	39% [32%, 47%]	40%
	Agreement	84% [74%, 92%]	85%	86% [77%, 94%]	86%
C^{read}	Coverage	63% [56%, 70%]	62%	55% [47%, 62%]	58%
	Agreement	81% [74%, 88%]	81%	83% [76%, 90%]	84%
C^{act}	Coverage	92% [88%, 96%]	91%	91% [86%, 95%]	91%
	Agreement	80% [74%, 86%]	81%	81% [75%, 87%]	81%
Unmet requirements labelled		observed / potential / unproven (judge A)			
C^{traj}		84 / 10 / 7			
C^{read}		224 / 55 / 8			
C^{act}		296 / 1 / 7			

Table D3 | Committed verdicts after the audit of named violations, on the 406 candidate final states. The paper's rule reproduces Table D1. Lenient and strict count a reject as committed only if an unmet requirement is labelled *observed* or *potential* (lenient) or *observed* (strict). Intervals are 95% bootstrap over tasks.

What the audit does not show. The judge reads the critic's report. It does not re-run the critic's checks, so it establishes that a named violation cites an observation, not that the observation is true. It also does not judge whether the critic reads the requirement as a reasonable reader would. Nor does it audit the requirements marked met, on which every accept rests. Agreement, in turn, is measured on verdicts, not on whether a named violation is the one that the verifier finds.

E The Critic as a Final Check

This appendix gives the protocol and results of the final check summarized in § 6.

Protocol. The actor works without knowing about the critic. When it calls `task_complete` at a state $S^{(1)}$, $C^{\text{act}}{+}S_0$ lists the task's requirements and checks each one by running commands in disposable forks of $S^{(1)}$ and S_0. If the critic accepts, the state goes to the verifier. Otherwise the actor receives the failed requirements, why they failed, and how the critic checked, and resumes from the unmodified $S^{(1)}$. The loop ends when the critic accepts, after six critic rounds, or when the time limit T runs out. If no state was accepted, the last proposed state is submitted.

Outcome. With GPT-5.6 Luna as actor and critic, Probe-and-Resume raises pass@1 from 63.4% to 72.4% ($+$9.0 points [$+$5.6, $+$12.6]; Table 1). It improves the pass rate on 30 of 89 tasks and lowers it on 1. It costs 2.2$\times$ as much as the baseline ($23.14 against $10.70), because the critic runs only when the actor believes it is done. The gain is similar with Claude Sonnet 5 or GLM-5 as the critic ($+$9.9 and $+$7.2 points) and on SWE-bench Pro (Deng et al., 2025) ($+$7.5 points, mostly on its medium tasks; Appendix F).

Error rates in the loop are close to the offline ones. Of the 136 first proposals that fail, the critic rejects 103 (76%), and the actor later fixes 43 of them (42%). Of the 282 passing proposals, the critic wrongly rejects 60 (21%), and 57 of these still end in a passing state. Xu et al. (2026a, Fig. 6a) traces every trial. The critic's false accept rate in the loop (21%) is close to the one measured offline for C^{act} (22%), although the final check uses $C^{\text{act}}{+}S_0$.

An error costs little when the actor can resume. A wrongly rejected actor resumes from the state it claimed, so only 3 of the 60 false rejects end as failures. Restarting the actor from S_0 after each rejection, with the same critic, feedback, and time limit, lowers success by 6.1 points (95% CI [$-$10.1, $-$2.2]) and costs 1.6$\times$ as much as resuming (Xu et al., 2026a, App. K). Withholding S_0 from the critic lowers it by 7.9 points [$-$11.9, $-$3.8], and letting the actor judge its own claims within its own conversation, instead of a fresh critic, lowers it by 7.2 points [$-$11.7, $-$3.1] (Table 1; paired bootstrap over tasks). The actor is a worse judge of its own work: of the claims it judges, it wrongly accepts 28.2% of the failing ones and wrongly rejects 36.3% of the passing ones, against 21.4% and 24.4% for the separate critic.

F Other Critics, Actors, and Benchmarks

This appendix repeats the final check of §6 with other critic models, with stronger models as the actor and the critic, and on SWE-bench Pro. Unless stated otherwise, the protocol is the one of Appendix E.

F.1 Different Critic Models

The critic does not have to use the actor's model. We keep GPT-5.6 Luna as the actor and use Claude Sonnet 5 or Z.AI's GLM-5 as the critic, with the same prompts, time limit T and number of critic rounds, on all 89 tasks (Table F1). The critic acts only after the first claim, so the first claims pass at about the same rate with every critic.

With Sonnet 5, the gain is as large as with Luna, and it comes about in the same way: the critic rejects 72% of the failing first claims, and the actor turns 49 of them into passing states. The per-task pass rate rises on 28 tasks and falls on 2. GLM-5 is a more lenient critic. It

rejects fewer failing first claims, so fewer go back to the actor, and also fewer passing ones (6% against 21% for Luna). Its gain is smaller than with the Luna critic, but the difference, -1.8 points $[-4.9, +1.6]$, is not significant.

Critic	First claims	Probe-and-Resume	Gain [95% CI]	Failing rejected	Passing lost
Luna	63%	72%	$+9.0$ points $[+5.6, +12.6]$	76%	3
Sonnet 5	61%	70%	$+9.9$ points $[+5.4, +14.6]$	72%	5
GLM-5	63%	70%	$+7.2$ points $[+4.3, +10.6]$	52%	1

Table F1 | Probe-and-Resume on Terminal-Bench 2.1 with GPT-5.6 Luna as the actor and each model as the critic (Luna: GPT-5.6 Luna; Sonnet 5: Claude Sonnet 5), five trials per task. *First claims*: pass rate of the actor's first claimed state (the baseline). *Failing rejected*: share of failing first claims that the critic rejects. *Passing lost*: passing first claims that end as failures after a false reject. Intervals are 95% bootstrap over tasks.

F.2 Stronger Models as Actor and Critic

GPT-5.6 Luna passes 63% of its first claims on Terminal-Bench 2.1. We repeat the loop with stronger models: GPT-6 Astra or Claude Opus 5.5 as the actor, and Claude Fable 5.1 or Opus 5.5 as the critic. We run them on 15 tasks chosen from the 31 tasks that Astra in the Codex CLI fails at least once in its 25 Harbor Hub trials: ten on which its first claim passes in some of our trials and fails in others, so that the loop has something to recover; four that it always passes, as controls for false rejections; and one that it never passes, as a control for false accepts. The other settings are unchanged.

None of the three combinations gains significantly (Table F2). With Luna, only 33% of the first claims on these tasks pass, and the loop works on the rest. With the stronger actors, most first claims already pass, so there is little to recover, and the remaining failures are ones that a critic of the same strength does not catch either. With Opus 5.5 on both sides, the critic accepts almost every first claim, and no trial is rescued or lost. Claude's content classifier answers two life-science tasks with empty replies, so some of the Opus 5.5 actor trials are refused; they count as failures.

Actor / Critic	Tasks	Trials	Baseline	Probe-and-Resume	Δ [95% CI]	Refused	Cost
Luna / Luna	15	75	33.3%	**40.0%**	$+6.7$ $[-2.7, +16.0]$	0	\$4.04
Astra / Fable	15	75	61.3%	**62.7%**	$+1.3$ $[-6.7, +9.3]$	0	\$126.94
Astra / Opus	15	75	69.3%	**73.3%**	$+4.0$ $[-2.7, +10.7]$	0	\$65.83
Opus / Opus	15	75	69.3%	69.3%	0.0 [0.0, 0.0]	8	\$52.80

Table F2 | Probe-and-Resume with stronger models as the actor and the critic, on the same 15 tasks of Terminal-Bench 2.1, five trials per task. Luna: GPT-5.6 Luna (the trials of Table 1 on these tasks); Astra: GPT-6 Astra; Fable, Opus: Claude Fable 5.1, Claude Opus 5.5. *Baseline* and Probe-and-Resume as in Table 1; Δ in points, with 95% bootstrap intervals over tasks. *Refused*: trials the actor's model refused; they count 0 in both columns. *Cost*: model cost of the trials that were not refused.

F.3 SWE-bench Pro

To test whether the final check generalizes beyond Terminal-Bench 2.1, we run Probe-and-Resume unchanged, with GPT-5.6 Luna as actor and critic, on the public split of SWE-bench Pro V2.0.0 (Deng et al., 2025). We use two task sets, both fixed by data released with the benchmark and independent of how Luna or Probe-and-Resume does: *Hard*, its HARD-51 subset, and *Medium*, the tasks that none of the benchmark's released runs of 2025 models solves. A task is scored only if its reference patch passes in our environment, which leaves

227 tasks (41 hard, 186 medium); we report one trial of each under a 50-minute limit, and four more trials of each hard task. As on Terminal-Bench 2.1, the actor and critic run offline except for the model endpoint, and each proposed state is graded by the task's unchanged verifier on a fork, so the hidden tests never reach the actor's or the critic's sandbox.

Probe-and-Resume raises pass@1 from 80% to 88% ($+7.5$ points [$+3.1$, $+11.9$]), similar to the gain on Terminal-Bench 2.1, and it costs 2.2$\times$ as much as the baseline. The critic rejects 36 of the 45 failing first claims (80%), and the actor turns 22 of them into passing states; it falsely rejects 46 of the 182 passing first claims (25%), and 41 of these still end in a passing state. The gain comes from the medium tasks; on the hard tasks it cannot be told apart from zero, with the first trial or with all five (Table F3).

	Tasks	Trials	Baseline	Probe-and-Resume	Gain [95% CI]
Hard, first trial	41	41	70.7%	73.2%	$+2.4$ [-4.9, $+9.8$]
Hard, all five trials	41	205	65.4%	66.3%	$+1.0$ [-3.9, $+5.9$]
Medium	186	186	82.3%	90.9%	$+8.6$ [$+3.8$, $+14.0$]
Both, first trial	227	227	80.2%	87.7%	$+7.5$ [$+3.1$, $+11.9$]

Table F3 | Probe-and-Resume on SWE-bench Pro by task set. Baseline: the first claimed state; Probe-and-Resume: the submitted state. Gains are in points, with 95% bootstrap intervals over tasks (Appendix C).

G Limitations

Critics help when actors can repair mistakes they fail to identify themselves; easy tasks the actor can solve leave little room for critics to help, while tasks that are too hard may not be solvable by the actor. As weak evidence, we find no statistically significant gains using GPT-6 Astra and Claude Opus 5.5 as actor or critic on even hard Terminal-Bench 2.1 tasks (Appendix F.2). Our evaluation manages environment state using Waypoint's terminal-oriented container. Other settings require branching support for their state, such as TClone for GUI workspaces (Huang et al., 2026). Forks do not isolate writes to external databases or shared services, so destructive checks require isolating those dependencies or restricting critic actions.