AUTOENVSCALING: AUTOMATING THE DATA FLYWHEEL WITH TERMINAL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Terminal agents solve tasks in software engineering and scientific research by executing commands in a terminal. Training them via reinforcement learning (RL) requires diverse and verifiable environments, but these environments are still mostly designed by human researchers, so their supply grows only as fast as researchers can work. We introduce **AUTOENVSCALING**, which treats environment design as a terminal task and uses agents to automate the data flywheel. A proposer agent builds new RL environments from the rollouts of the solver being trained, and the solver's new rollouts guide the next round. By turning environment generation into a terminal task, AUTOENVSCALING increases the valid-environment rate by up to $25\times$ compared to prompt-only generation, at up to 75% lower cost per valid environment. We show that training Qwen3.5-35B-A3B on AUTOENVSCALING environments from GPT-5.6-sol improves its Terminal-Bench 2.1 score by 4.3 points over the matched Tmax baseline. The broader goal of AUTOENVSCALING is to enable recursive self-improvement (RSI): by making both environment generation and solving terminal tasks, it lets one model co-evolve in both roles. We show that Qwen3.6-35B-A3B, acting in both roles, gains 12.2 points on Terminal-Bench 2.1 and raises its Terminal-Bench 4.0 score from 1.6 to 3.2. The pipeline also generalizes across agentic domains, generating environments for software engineering, computer use, professional work, and GPU kernel programming. Overall, these results show a path toward agents building their own training environments, reducing human reliance and taking a core step toward RSI.

1 INTRODUCTION

Terminal agents now solve tasks across diverse domains, from software engineering and scientific research to data analysis and cybersecurity (Jimenez et al., 2024; Terminal-Bench-Science Team, 2026; Wang et al., 2025a), by running in command-line environments (Merrill et al., 2026). Reinforcement learning (RL) has become the primary method for improving these agents (Wei et al., 2025a; Wang et al., 2025b; Jin et al., 2025), and they depend on diverse, high-quality training environments. Building these environments has become a major industry investment, with companies reportedly considering annual spending exceeding $1 billion on RL environments (Zeff, 2025). However, high-quality environments are scarce and lose their value once the agent has learned what they teach. As an agent improves, researchers need to re-diagnose which skills it still lacks and design new environments for further RL training. Agent training is therefore bottlenecked by human effort: the supply of useful environments grows only as fast as researchers can work.

Several lines of work reduce this human effort by having LLMs generate the environments, each with its own limitation. *Pipeline synthesis* runs LLMs through a pre-defined pipeline, such as a taxonomy of domains and skills or a fixed sequence of generation steps, to scale environments (Wang et al., 2026b; Tu et al., 2026; Zhang et al., 2026d; Chen et al., 2026; Ivison et al., 2026; Gandhi et al., 2026; Zhu et al., 2026). These pools are fixed before training and do not adapt as the agent improves. *Feedback-driven methods* add a feedback loop from the environment: they run the generated tasks and repair, extend, or adjust the difficulty of tasks based on the feedback (Shen et al., 2026; Li et al., 2026b; Fan et al., 2026; Yang et al., 2026; Wu et al., 2026a; Cheng et al., 2026), but how the feedback is used is still hand-designed. *Agentic methods* hand the work to a coding agent: AutoData creates QA data guided by solver outputs (Kulikov et al., 2026), and EnvHarness customizes existing environments (Huang et al., 2026), but neither builds new environments. *Self-play* trains an LLM

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

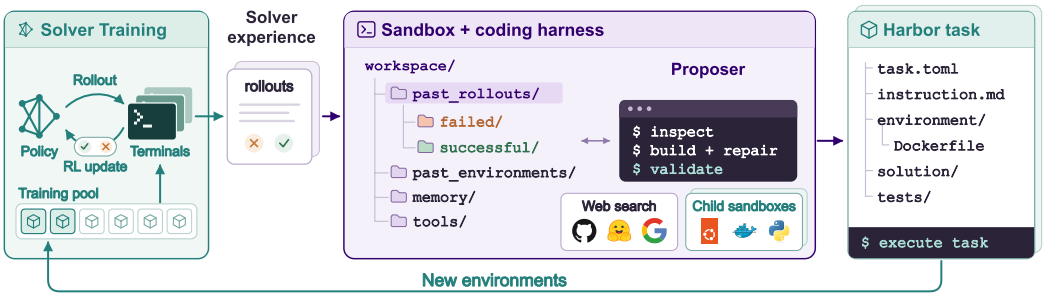


Figure 1: **AUTOENVSCALING automates the data flywheel.** The proposer works inside a sandbox with a coding harness, solver rollouts, a workspace, and web access to construct Harbor training environments. It can launch child sandboxes to build and test tasks. The solver trains on accepted environments with verifier rewards; its new rollouts guide the next round of environment generation.

as a proposer that generates harder tasks as the solver improves (Zhao et al., 2025; Huang et al., 2025; Liu et al., 2025b; 2026), but the proposer still generates tasks through a fixed pipeline, much like pipeline synthesis. In all of these methods, researchers fix how environments are generated in advance. We instead let an agent carry out the researcher's workflow end to end: it reads the solver's rollouts, decides what to build next, builds and tests complete environments in a terminal, and improves at this job across rounds through its harness or weights.

We introduce **AUTOENVSCALING**, which treats environment design as a terminal task: building an environment means writing and debugging code, the same work terminal agents are trained to do. With access to the solver's rollouts, online search, and a sandbox, a proposer agent follows a researcher's workflow and builds complete Harbor tasks (Harbor Framework Team, 2026), each with instructions, an execution environment, a reference solution, and tests. Each environment is validated and calibrated against the current solver before it enters the training pool, and the solver's new rollouts guide the next round, forming a data flywheel (Figure 1). The proposer can also co-evolve with the solver across rounds, through harness optimization or self-play.

We first compare AUTOENVSCALING with pipeline synthesis across eight frontier models. It raises the valid-environment rate by up to $25\times$ and lowers the cost per valid environment by up to 75%. Environments designed by GPT-5.6-sol and Claude Opus 5 improve Qwen3.5-9B and Qwen3.5-35B-A3B by up to 7.4 and 4.3 points on Terminal-Bench 2.1 over static Tmax pools, and the gain shrinks without solver feedback. Finally, we show that a model can design its own training environments: with Qwen3.6-35B-A3B as both proposer and solver, AUTOENVSCALING improves Terminal-Bench 2.1 by 12.2 points over the Cold-Start checkpoint and raises Terminal-Bench 4.0 from 1.6 to 3.2. The environment design that researchers do by hand can thus be taken over by the agent being trained, a step toward recursive self-improvement (RSI).

Our contributions are as follows:

1. We introduce a **framework that turns environment design into a terminal task**, where a proposer agent decides what to build, then builds and tests environments inside a sandbox. The same setup extends to software engineering, computer use, professional work, and GPU kernel tasks.
2. We design a **solver-guided data flywheel for terminal RL**, which validates and calibrates new environments against the solver, so the training pool changes as the solver improves. Frontier models can serve as the proposer with harness improvement across rounds.
3. We show **recursive self-improvement with a single policy**: the same model acts as both proposer and solver, training on its own designed environments and using its updated weights to design the next round. Training the proposer keeps this loop stable and lets both roles co-evolve.

2 RELATED WORK

Environment generation. Table 1 compares generation strategies, data sources, and proposer learning. Endless Terminals, TermiGen, and Tmax generate executable terminal tasks (Gandhi et al., 2026; Zhu et al., 2026; Ivison et al., 2026), while SETA adds container, oracle, and no-op checks (Shen et al., 2026). SWE-Universe and Terminal-Universe reconstruct environments from

Table 1: **Comparison of environment generation methods.** 'External' uses a separate proposer model; 'Self' uses the trained solver as proposer. Proposer updates distinguish harness optimization from weight training. Entries describe the main settings, excluding ablations.

Method	Strategy	Data Sources	Proposer Source		Proposer Optimization	
			External	Self	Harness	Weights
Endless Term. (Gandhi et al., 2026)	Prompt + feedback	Human-defined	✓	×	×	×
TermiGen (Zhu et al., 2026)	Prompt + feedback	Human-defined	✓	×	×	×
Terminal-World (Cheng et al., 2026)	Prompt + feedback	Skill-based	✓	×	×	×
Tmax (Ivison et al., 2026)	Prompt	Human-defined	✓	×	×	×
SETA (Shen et al., 2026)	Harness	Seed tasks, human-defined	✓	×	×	×
RST (Li et al., 2026b)	Prompt + feedback	Seed tasks	✓	×	×	×
Env. Evolution (Fan et al., 2026)	Prompt + feedback	Seed tasks	✓	×	×	×
Terminal-Uni (Wu et al., 2026a)	Prompt + feedback	Solver rollouts	✓	×	×	×
T1 (Yang et al., 2026)	Prompt + feedback	Seed tasks	✓	×	×	×
AutoData (Kulikov et al., 2026)	Harness	Corpus, solver rollouts	✓	×	✓	×
SPADE (Liu et al., 2026)	Prompt	Corpus, memory	×	✓	×	✓
AUTOENVSCALING (Ours)	Harness	Corpus, rollouts, memory, web	✓	✓	✓	✓

repository changes or execution histories (Chen et al., 2026; Wu et al., 2026a). RST and Environment Evolution expand verified tasks, and T1 trains on a fixed synthesized pool (Li et al., 2026b; Fan et al., 2026; Yang et al., 2026). Concurrent work also calibrates tasks against a pool of solvers (Meng et al., 2026) or rewrites seed tasks toward a target pass rate (Wu et al., 2026b), but the generator itself does not learn from its results. AUTOENVSCALING instead places a proposer inside the training loop to build and calibrate new environments against the current solver, so the training pool evolves as the solver improves.

Adaptive curricula and self-play. Curriculum learning and unsupervised environment design adapt training tasks to the learner (Bengio et al., 2009; Schmidhuber, 2013; Dennis et al., 2020). AutoData revises data-generation prompts using solver outputs (Kulikov et al., 2026); EnvHarness edits environment wrappers while preserving verifiers (Huang et al., 2026). Proposer–solver self-play couples task generation and learning (Zhao et al., 2025; Liu et al., 2025b; Xia et al., 2025). Self-Challenging agents write tasks together with a verifier, a solution, and failure cases that filter them (Zhou et al., 2025), and SSR trains one agent to inject and repair bugs in sandboxed repositories (Wei et al., 2025b). SPADE trains a shared model to design and solve Gym-style environments, using a hint-based regret reward for the designer (Liu et al., 2026). AUTOENVSCALING additionally improves the proposer's harness, retaining tools, skills, and memory across rounds, so closed-weight proposers can also improve.

Recursive self-improvement. Recursive self-improvement (RSI) describes AI systems that help build more capable successors, which in turn improve the next generation (Anthropic Institute, 2026; OpenAI, 2026). Existing work pursues this by improving what surrounds the model: agent harnesses (Hu et al., 2025; Zhang et al., 2026c; Wang et al., 2026a; Karten et al., 2026; Zhang et al., 2026b; Lee et al., 2026), training algorithms and infrastructure (Oh et al., 2025; Novikov et al., 2025), inference systems and GPU kernels (Xing et al., 2026; Ouyang et al., 2025; Yuksekgonul et al., 2026), chip designs (Mirhoseini et al., 2021), and ML research (Si et al., 2026; Lyu et al., 2026). AUTOENVSCALING targets the training data: we train the model to design better environments for its own training, so what it learns as a solver can improve the data it generates next.

3 AUTOENVSCALING: AUTOMATING THE DATA FLYWHEEL WITH TERMINAL AGENTS

AUTOENVSCALING makes environment design a terminal task: the proposer agent builds training environments by exploring its workspace, writing environments, and running and fixing tests inside a sandbox (Figure 2). Like a human researcher, it reads the solver's rollouts to see which tasks

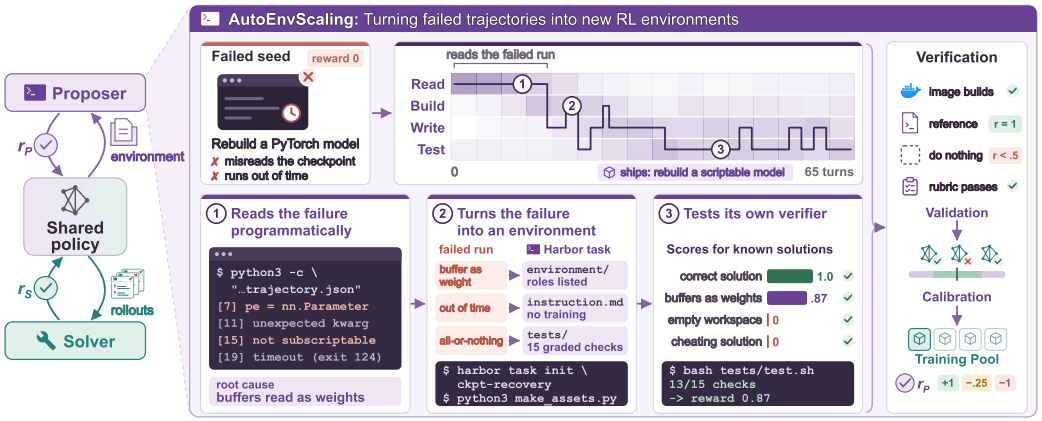


Figure 2: **Environment design and solving as terminal tasks. Left**: one shared policy acts as both proposer and solver. As proposer it designs environments and earns r_P (Equation 2), which depends on whether each environment passes validation and calibration. As solver it attempts tasks and earns r_S (Equation 1), the fraction of each task's tests it passes. **Right**: an example of the proposer turning the failed PyTorch seed from Section 4.4 into a new environment. The trace follows its read, build, write, and test actions, and the shading shows where its runs spend their turns.

the solver fails and which it solves, and designs new environments to match (Section 3.1). An environment enters the training pool only after validation and calibration (Section 3.2). The solver trains on this pool with RL, and its new rollouts feed the next round, while the proposer improves its harness or weights (Section 3.3).

3.1 Turning Environment Design into a Terminal Task

Each proposer episode produces one environment as a complete Harbor task (Harbor Framework Team, 2026),[1] with an instruction, an execution environment, a reference solution, and tests. The solver sees only the instruction and works inside the environment's container, while the reference solution and tests stay hidden. The proposer works in a sandbox, an isolated container with a workspace, child containers for building and testing environments, and restricted web access. We call the sandbox together with the proposer's skills, tools, and memory the proposer's harness.

Workspace. The workspace contains the solver's past rollouts with their verifier outputs, previously generated environments, instructions for writing Harbor tasks, and the proposer's own memory and tools. The rollouts and past environments are refreshed from the training pool each round, while the memory and tools persist across rounds. The proposer can edit its memory and tools but not the instructions or the admission checks, which run outside the sandbox. It can also search the web for real problems, documentation, and data, while egress rules block Terminal-Bench and other benchmark pages.

Assignments from solver rollouts. Each episode starts from an assignment that names a parent environment from the training pool and gives the solver's pass rate and most common failure on it. The proposer reads the solver's trajectories and verifier outputs on the parent to find what went wrong or what the solver already does reliably. It then builds a simpler variant when the solver keeps failing, or a harder one with a longer horizon or an additional skill when the solver succeeds. Figure 2 shows an example: after reading a failed PyTorch checkpoint-recovery run, Claude Opus 5 found that the solver had loaded buffers as weights and built a new environment around that mistake.

End-to-end design. Prior generators fix the stages, sampling axes, or rewrite operators of generation in advance (Ivison et al., 2026; Shen et al., 2026; Yang et al., 2026). We instead let the proposer decide what to build and when the environment is finished. It follows Harbor's official `create-task` skill, which walks through each file of a Harbor task and ends with a check by

[1] `https://www.harborframework.com/docs/tasks`

Harbor's oracle agent. The proposer writes the environment's files in the workspace, launches child containers that build its image and run its reference solution and tests, and revises the environment until the tests behave as intended. It can also run the current solver on a new environment to estimate its difficulty. Appendices D and E give the workspace layout, sandbox resources, and skills.

3.2 Environment Validation and Calibration

Before a new environment enters the training pool, the host checks that it works and that it suits the current solver.

Validation. The host builds the environment and runs its tests on two solutions: the proposer's reference solution, which should receive full reward, and an empty solution, which should receive less than half. We also perform a contamination check by flagging any instruction that shares a 13-gram with a held-out task, following Tmax (Ivison et al., 2026). These checks cannot catch a mistake that the reference solution and the tests share, since the proposer wrote both, nor tests that reward a shortcut. For these cases we run Harbor's task review (`harbor check`), in which the proposer's model reviews the task against the official Terminal-Bench rubric.

Calibration. An environment that passes validation is then attempted several times by the current solver. We keep it if the mean reward falls between 0.25 and 0.75 and the rewards vary across attempts, so that the solver sometimes solves it and sometimes does not. The proposer can run this check itself before submitting, using a `solver_model` tool similar to the ones in AutoData and GPT-Red (Kulikov et al., 2026; Wallace et al., 2026). Environments that fail validation or calibration go back to the proposer for revision (Appendix G.4).

3.3 Training the Proposer and Solver

Admitted environments form the solver's training pool. At each refresh, the host reruns the updated solver on the pool, removes environments that fall outside the calibration band, and asks the proposer for replacements based on the latest rollouts.

Agent trajectories and rewards. Both roles act in a terminal in the same way. Given a prompt x and a container C, the policy issues a command or tool call $a_t \sim \pi_\theta(\cdot \mid x, a_{<t}, o_{<t})$ and reads its output o_t from C, producing a trajectory $\tau = (x, a_1, o_1, \ldots, a_T, o_T)$. The two roles differ only in the prompt, the container, and how the trajectory is scored. The solver starts from an environment's instruction inside that environment's container, and its trajectory τ_S is scored by the fraction of the environment's N tests it passes:

$$r_S(\tau_S) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[\text{test } i \text{ passes}], \tag{1}$$

so a partial solution receives partial reward. The proposer starts from an assignment and its skills inside its sandbox, and its trajectory τ_P ends with a new environment, which is scored by

$$r_P(\tau_P) = \begin{cases} -1, & \text{if validation fails,} \\ -0.25, & \text{if validation passes but calibration fails,} \\ 1, & \text{if both pass.} \end{cases} \tag{2}$$

This reward separates broken environments from working ones that do not suit the current solver, and it gives full reward only to environments that enter the training pool.

Harness optimization. Following Continual Harness (Karten et al., 2026), the proposer updates its own memory, skills, and tools between rounds based on its past rewards and logs. For example, it merged several validation steps into a faster script and saved useful GitHub tasks and Docker Hub images as a web-cache skill. The validation and calibration checks stay fixed.

Policy updates. When the proposer has open weights, one policy plays both roles and we train it on both rewards. We use Divergence Proximal Policy Optimization (DPPO) (Qi et al., 2026) with

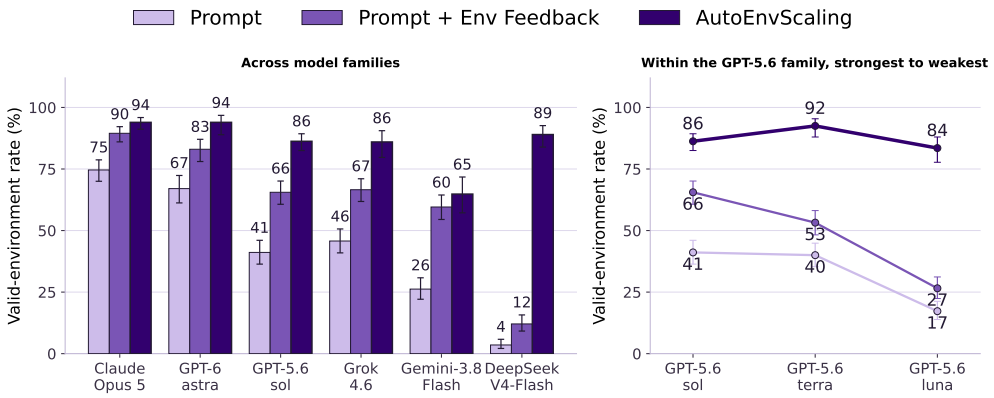


Figure 3: **AUTOENVSCALING closes the gap between weaker and stronger models.** Valid-environment rate across eight proposers. **Left:** Prompt $<$ Prompt + Env Feedback $<$ AUTOENVSCALING for every model. **Right:** Within GPT-5.6, where Sol is the strongest model and Luna the weakest, prompt-based rates fall as the model weakens, while ours stay between 84% and 92%.

separate reward groups for the two roles. For role $q \in \{P, S\}$, let $\tau_{q,1}, \ldots, \tau_{q,G_q}$ be a group of trajectories with rewards $r_{q,i} = r_q(\tau_{q,i})$. The advantage of $\tau_{q,i}$ is

$$\hat{A}_{q,i} = r_{q,i} - \frac{1}{G_q} \sum_{j=1}^{G_q} r_{q,j}. \tag{3}$$

Jointly training the proposer and solver introduces challenges common to multi-agent RL: their rewards differ, and the environments change as both improve. Following previous self-play work (Liu et al., 2025a; 2026), we compute advantage baselines separately for the two roles. Proposer rewards are centered within proposer groups, while solver rewards are centered across attempts on the same environment. Following DAPO (Yu et al., 2025), we drop groups whose rewards do not vary. The solver trains at every step. Every k solver steps, the proposer generates new environments and trains on the resulting proposer trajectories. Further details are given in Appendix G.

4 EXPERIMENTS

We first test the core hypothesis behind AUTOENVSCALING, that treating environment design as a terminal task yields more valid environments at lower cost than prompting (Section 4.1). We then show how AUTOENVSCALING lets frontier models design better environments for RL training (Section 4.2). Next, we show how unifying both roles as terminal tasks allows AUTOENVSCALING to move beyond frontier-model proposers and enable recursive self-improvement, with a single policy training and co-evolving as proposer and solver (Section 4.3).

4.1 DOES ENVIRONMENT DESIGN BENEFIT FROM A TERMINAL?

Setup. We first study why environment generation should run as a terminal task. We compare three methods from Table 1: **(1) Prompt**, representing pipeline synthesis, asks the model to generate environment files from a task description, without executing them during generation (Ivison et al., 2026; Liu et al., 2026); **(2) Prompt + Env Feedback**, representing feedback-driven methods, runs the generated environment and returns build and verifier logs to the model for revision, following RST (Li et al., 2026b) and Environment Evolution (Fan et al., 2026); **(3) AUTOENVSCALING** is our harness-based generation method. It places the model directly inside a sandbox, where it inspects files, runs commands, and validates and repairs environments as terminal tasks. We evaluate all three methods on eight frontier models, including GPT-6-astra, Claude Opus 5, Gemini-3.8-Flash, Grok 4.6, and DeepSeek-V4-Flash. We count an environment as valid if it builds, its reference solution passes the tests, and an empty solution fails them. For each method, we report the percentage of generated environments that are valid and the cost per valid environment.

Table 2: **Valid-environment rate and cost across three generation methods.** Costs are USD per valid environment. Appendix F examines reference execution during generation.

	Valid-env. rate (%)			Cost per valid env. (USD)		
Proposer	Prompt	Prompt + Env Feedback	AUTOENVSCALING	Prompt	Prompt + Env Feedback	AUTOENVSCALING
Claude Opus 5	74.6 [70.0, 78.7]	89.5 [86.0, 92.2]	**94.0** [91.2, 95.9]	0.72	0.70	$1.18_{1.6\times\uparrow}$
GPT-6-astra	67.0 [61.2, 72.4]	83.0 [78.0, 87.1]	**94.0** [88.9, 96.8]	1.79	1.76	$2.75_{1.5\times\uparrow}$
GPT-5.6-sol	41.1 [36.4, 46.0]	65.5 [60.7, 70.1]	**86.3** [82.5, 89.3]	0.90	0.80	$0.59_{0.7\times\downarrow}$
GPT-5.6-terra	40.0 [35.3, 44.9]	53.2 [48.2, 58.1]	**92.5** [88.0, 95.4]	0.13	0.15	$0.09_{0.7\times\downarrow}$
Gemini-3.8-Flash	26.2 [22.1, 30.8]	59.6 [54.5, 64.4]	**64.8** [57.2, 71.7]	0.28	0.20	$0.16_{0.6\times\downarrow}$
GPT-5.6-luna	17.2 [13.9, 21.3]	26.5 [22.4, 31.2]	**83.5** [77.7, 88.0]	0.02	0.03	$0.01_{0.4\times\downarrow}$
DeepSeek-V4-Flash	3.5 [2.1, 5.8]	12.1 [9.2, 15.7]	**89.0** [83.9, 92.6]	0.08	0.06	$0.02_{0.25\times\downarrow}$
Grok 4.6	45.8 [40.9, 50.6]	66.6 [61.8, 71.0]	**86.0** [79.7, 90.6]	0.59	0.69	$0.44_{0.7\times\downarrow}$

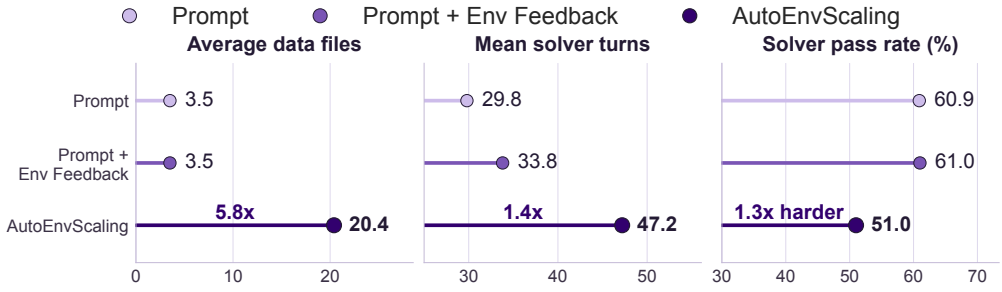


Figure 4: **Environments generated by AUTOENVSCALING are richer, longer-horizon, and harder to solve.** Files per accepted environment, solver turns, and solver pass rate for each method.

Results. Figure 3 shows that AUTOENVSCALING achieves the highest valid-environment rate across all eight models. The largest gain is for DeepSeek-V4-Flash, from 3.5% with Prompt to 89.0% with AUTOENVSCALING. These gains support the "mismanaged geniuses" hypothesis (Zhang et al., 2026a): models can be held back by scaffolds that fail to make full use of their capabilities. Figure 3 (right) shows that prompt-based generation degrades across Sol, Terra, and Luna as model capability decreases, while AUTOENVSCALING maintains high valid-environment rates. Table 2 shows that AUTOENVSCALING lowers cost per valid environment for six of the eight models, with reductions of up to 75% compared with Prompt.

Figure 4 examines the generated environments quantitatively. AUTOENVSCALING produces tasks with more data files and longer solver trajectories. Compared with Prompt + Env Feedback, the average number of solver turns increases from 33.8 to 47.2, while the solver pass rate falls from 61.0% to 51.0%. Appendix C shows that AUTOENVSCALING also generates environments for software engineering, computer use, professional work, and GPU kernel programming.

4.2 AUTOENVSCALING'S MODEL-DESIGNED CURRICULA OUTPERFORM STATIC ENVIRONMENT POOLS

Training setup. We train Qwen3.5-9B and Qwen3.5-35B-A3B as solvers with GPT-5.6-sol or Claude Opus 5 as the proposer. These proposers keep their weights fixed and improve only through their harness, and RL updates the solver. We use a group size of 16 and 16 prompts per batch, with a maximum sequence length of 128k tokens. Full training and evaluation settings are in Appendix G.

Baseline and Evaluation. For a fair comparison, we regenerate the Tmax pool using GPT-5.6-sol and Claude Opus 5, following the Tmax recipe (Ivison et al., 2026). Each comparison therefore uses the same proposer for the static pool and AUTOENVSCALING. We evaluate on held-out Terminal-Bench 2.1 (Merrill et al., 2026), reporting avg@5 (Appendix G.3).

Results. Table 3 shows that AUTOENVSCALING outperforms static Tmax pools with both frontier proposers.

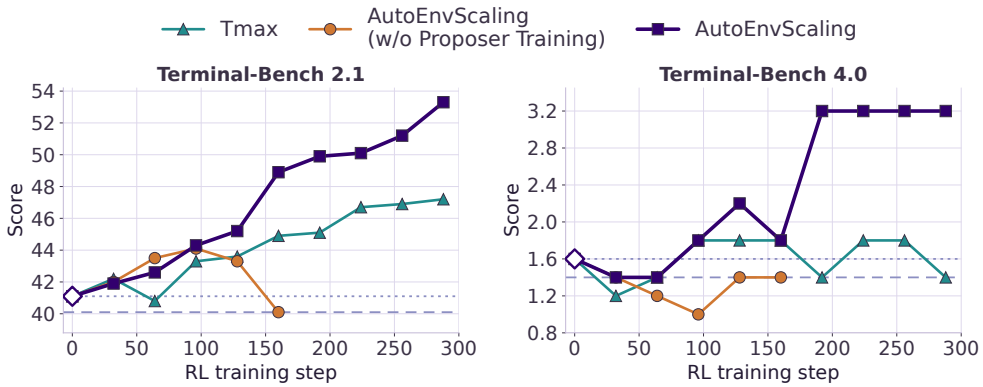


Figure 5: **AUTOENVSCALING enables recursive self-improvement.** Qwen3.6-35B-A3B performance on Terminal-Bench 2.1 (left) and 4.0 (right) during training. We compare AUTOENVSCALING, AUTOENVSCALING without proposer training, and static Tmax training. The run without proposer training ends earlier because training collapsed.

For Qwen3.5-9B, environments generated by GPT-5.6-sol and Claude Opus 5 improve Terminal-Bench 2.1 scores to 33.5 and 36.6, respectively, exceeding the corresponding static pools by 6.3 and 7.4 points. For Qwen3.5-35B-A3B, AUTOENVSCALING reaches 38.9 with GPT-5.6-sol and 39.3 with Claude Opus 5, outperforming the matched Tmax baselines by 4.3 and 3.8 points. We also performed an ablation by removing past solver trajectories from the proposer's workspace. Qwen3.5-35B-A3B's score then drops by 3.6 points with GPT-5.6-sol and 6.7 points with Claude Opus 5, leaving it 0.7 points above Tmax with GPT-5.6-sol and 2.9 points below with Claude Opus 5 (Table 3). This suggests that solver feedback helps the proposer build a more effective training curriculum. As the solver improves, its new failures guide the proposer toward tasks that remain challenging, allowing the curriculum to grow with the solver's capabilities.

Table 3: **AUTOENVSCALING versus baseline.** Held-out Terminal-Bench 2.1 under matched training steps. $\pm$ gives the confidence interval.

	Proposer	
Model / Settings	**GPT-5.6-sol**	**Claude Opus 5**
Qwen3.5-9B	19.1 $\pm$ 2.3	
+ Tmax	27.2 $\pm$ 1.4	29.2 $\pm$ 2.1
+ AUTOENVSCALING	**33.5 $\pm$ 1.9**	**36.6 $\pm$ 1.7**
Qwen3.5-35B-A3B	26.7 $\pm$ 2.2	
+ Tmax	34.6 $\pm$ 1.9	35.5 $\pm$ 1.6
+ AUTOENVSCALING (w/o solver feedback)	35.3 $\pm$ 2.3	32.6 $\pm$ 1.7
+ AUTOENVSCALING	**38.9 $\pm$ 2.4**	**39.3 $\pm$ 1.6**

4.3 AUTOENVSCALING ENABLES RECURSIVE SELF-IMPROVEMENT

We test whether AUTOENVSCALING can keep improving a model without a stronger external proposer, as a step toward recursive self-improvement. We use Qwen3.6-35B-A3B as both proposer and solver: it designs training environments, trains on them, and uses the updated checkpoint for the next generation round. Prior self-play methods for LLMs give the two roles different kinds of tasks (Huang et al., 2025; Liu et al., 2025b; Yu et al., 2026; Liu et al., 2026). In AUTOENVSCALING, both roles are given terminal tasks, so what the model learns as solver can carry over to designing its next environments.

Settings. We follow the settings in Section 4.2, with Qwen3.6-35B-A3B as both proposer and solver. The base model generates valid environments only 32.3% of the time, too rarely to start training from. We therefore cold-start it by fine-tuning on 600 high-quality proposer trajectories. The resulting Cold-Start checkpoint raises the valid-environment rate to 86.8%, comparable to frontier models (Appendix H). We compare AUTOENVSCALING with training Qwen3.6-35B-A3B on the static Tmax pool, starting both from the same Cold-Start checkpoint. We evaluate on Terminal-Bench 2.1 and 4.0 (Merrill et al., 2026; Terminal-Bench Team, 2026).

Results. Figure 5 shows that Qwen3.6-35B-A3B improves by training on its own environments. On Terminal-Bench 2.1, AUTOENVSCALING raises avg@5 from 41.1 to 53.3, a 12.2-point gain and 6.1 points above Tmax. Terminal-Bench 4.0 is far harder for this model, which starts at 1.6. Its

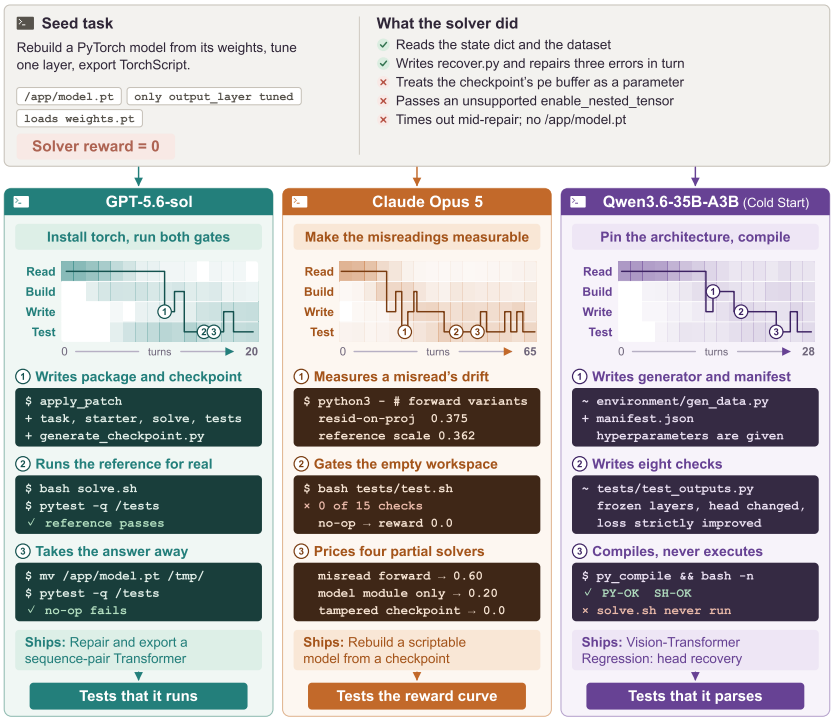


Figure 6: **Environment design from the same base-agent failure.** The base agent timed out rebuilding a PyTorch model from its weights. Each line follows one proposer through Read, Build, Write, and Test actions. Numbered examples show the corresponding generation and validation actions. Additional cases and the annotation protocol are in Appendix I.

score rises by 1.6 points to 3.2, compared with 1.4 for Tmax, even though the curriculum was not designed for Terminal-Bench 4.0. When we keep the weights shared but remove the proposer-side training, the model improves at first and then declines as environment validation errors grow and destabilize training (Figure 19), which motivates the proposer reward r_P.

We also study the training dynamics (Figure 18, Appendix G). AUTOENVSCALING provides a curriculum better suited to the improving solver, with a $3.1\times$ higher usable-group rate than Tmax. As a result, collecting rollouts for each training update takes 42% less time on average.

4.4 QUALITATIVE STUDY FOR AUTOENVSCALING

Figure 6 compares how GPT-5.6-sol, Claude Opus 5, and Qwen3.6-35B-A3B (Cold-Start) from Section 4.3, generate environments from the same failed solver run, in which the base agent timed out rebuilding a PyTorch model from its weights. The Read/Build/Write/Test plots show that Claude Opus 5 repeatedly alternates between writing and testing, while GPT-5.6-sol and Qwen3.6-35B-A3B follow shorter sequences. GPT-5.6-sol tests the reference solution and an empty workspace, while Claude Opus 5 additionally checks the rewards assigned to four partial solutions. Qwen3.6-35B-A3B (Cold-Start) writes the task files but only checks compilation in this example.

5 CONCLUSION

AUTOENVSCALING turns environment design into a terminal task by placing the proposer inside a sandbox with a coding harness, where it builds and validates training environments from the solver's past failures. Unifying environment design and solving as terminal tasks allows a single model to construct its own curriculum and learn from it. The proposer improves its harness and weights across training rounds, while the solver learns from the environments it generates. Our results show that this evolving curriculum outperforms fixed environment pools, taking a step toward recursive self-improvement and automating the data flywheel for post-training. We believe this pipeline could make the design of training environments an ability that agents learn and improve themselves.

AI USE STATEMENT

Language models are part of the method: they generate executable environments, reference solutions, and tests, and they act as solvers within the generated environments. The programmatic validation checks and the calibration against the current solver are described in Section 3.2 and Appendix G. We also use the proposer's own model to review each environment against the Terminal-Bench task rubric; these reviews are model judgments, not independent human annotations. AI assistants were used to discuss experiment design, interpret results, find related work, write plotting code, and prepare figures. The authors remain responsible for the manuscript and its claims.

ETHICS STATEMENT

AUTOENVSCALING studies how terminal agents can generate their own training environments. Stronger terminal agents have beneficial uses in software engineering and data analysis, but can also support harmful automation, for example in cybersecurity. Benchmark gains do not establish safe behavior in deployment. Generated environments and verifiers may contain errors, exploitable rewards, or unsafe code; our validation checks and rubric review reduce these risks but are not a security audit. We run generation and training in isolated sandboxes, with egress rules that block benchmark pages exposing evaluation tasks, and anyone reusing the pipeline should likewise isolate generated code, restrict permissions, and avoid exposing credentials or sensitive data. Web search can carry biases, copyrighted material, or personal information into generated environments, so generated data require review before release or reuse. Finally, repeated generation, rollout, and training incur computational and environmental costs; Appendix G reports the hardware and training settings so readers can assess them.

REPRODUCIBILITY STATEMENT

Section 3 specifies the environment-design workflow, the validation and calibration checks, and the training objectives, and Section 4 describes the models, baselines, training settings, and evaluation protocol. Appendix D describes the proposer's workspace, sandbox resources, and coding agents, and Appendix E reproduces both skills given to the proposer. Appendix F documents reference execution during generation and decontamination, Appendix G gives the training and calibration settings, and Appendix H reports the Cold-Start controls. These descriptions distinguish the programmatic checks from the model-based rubric review disclosed above.

REFERENCES

Pranjal Aggarwal, Graham Neubig, and Sean Welleck. Gym-Anything: Turn any software into an agent environment. *arXiv preprint arXiv:2604.06126*, 2026. URL https://arxiv.org/abs/2604.06126.

Anthropic Institute. When AI builds itself. https://www.anthropic.com/institute/recursive-self-improvement, 2026. Accessed 2026-09-22.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 41–48. ACM, 2009. doi: 10.1145/1553374.1553380.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, and Binyuan Hui. SWE-Universe: Scale real-world verifiable environments to millions, 2026. URL https://arxiv.org/abs/2602.02361.

Zihao Cheng, Hongru Wang, Zeming Liu, Xinyi Wang, Xiangrong Zhu, Yuhang Guo, Wei Lin, Jeff Z. Pan, and Yunhong Wang. Terminal-World: Scaling terminal-agent environments via agent skills. *arXiv preprint arXiv:2605.20876*, 2026. URL https://arxiv.org/abs/2605.20876.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. *Advances in Neural Information Processing Systems*, 33:13049–13061, 2020.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiang Zhou, Jiangtao Guan, Jincheng Liu, Yun Yang, Dingxin Hu, Zhuo Han, Xing Wu, Feng Zhang, and Lilin Wang. Environment Evolution for Terminal Agents, 2026. URL https://arxiv.org/abs/2609.04128.

Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents. *arXiv preprint arXiv:2601.16443*, 2026.

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

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-Zero: Self-evolving reasoning LLM from zero data. *arXiv preprint arXiv:2508.05004*, 2025.

Chengsong Huang, Zifeng Wang, Rujun Han, Jun Yan, Yanfei Chen, Zoey CuiZhu, Ke Jiang, Peng Xia, Han Yu, Yufan Zhuang, Yifei Ming, Jiaqi Pan, Bhavana Dalvi Mishra, Jiaxin Huang, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. EnvHarness: Awakening static worlds for agent learning, 2026. URL https://arxiv.org/abs/2608.19880.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026.

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.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-R1: Training LLMs to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Seth Karten, Joel Zhang, Tersoo Upaa Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*, 2026.

Ilia Kulikov, Chenxi Whitehouse, Tianhao Wu, Yixin Nie, Swarnadeep Saha, Eryk Helenowski, Weizhe Yuan, Olga Golovneva, Jack Lanchantin, Yoram Bachrach, Jakob Foerster, Xian Li, Han Fang, Sainbayar Sukhbaatar, and Jason Weston. Autodata: An agentic data scientist to create high quality synthetic data. *arXiv preprint arXiv:2606.25996*, 2026.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, Junteng Liu, Zhaochen Su, Yiyang Guo, Fan Zhou, Lueyang Zhang, Juan Michelini, Xingyao Wang, Xiang Yue, Shuyan Zhou, Graham Neubig, and Junxian He. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. In *International Conference on Learning Representations*, 2026a. URL https://arxiv.org/abs/2510.25726.

Zhongzhi Li, Yucheng Shi, Zongxia Li, Ruhan Wang, Anhao Li, Zixun Huang, Junyao Yang, Lei Ke, Ninghao Liu, Haitao Mi, and Leowei Liang. Recursive synthesis for long-horizon terminal tasks. *arXiv preprint arXiv:2608.05466*, 2026b. Tencent HY LLM Frontier.

Bo Liu, Leon Guertler, Simon Yu, Zichen Liu, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, Wee Sun Lee, and Natasha Jaques. Spiral: Self-play on zero-sum games incentivizes reasoning via multi-agent multi-turn reinforcement learning. *arXiv preprint arXiv:2506.24119*, 2025a.

Bo Liu, Chuanyang Jin, Seungone Kim, Weizhe Yuan, Wenting Zhao, Ilia Kulikov, Xian Li, Sainbayar Sukhbaatar, Jack Lanchantin, and Jason Weston. Spice: Self-play in corpus environments improves reasoning. *arXiv preprint arXiv:2510.24684*, 2025b.

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, Mickel Liu, Hanfei Yu, Zhaorun Chen, Weiyan Shi, Paul Pu Liang, Luke Zettlemoyer, Yejin Choi, and Natasha Jaques. SPADE: Self-play in adaptive synthetic executable environments, 2026. URL https://arxiv.org/abs/2608.19197.

Bohan Lyu, Yucheng Yang, Siqiao Huang, Jiaru Zhang, Qixin Xu, Xinghan Li, Xinyang Han, Yicheng Zhang, Huaqing Zhang, Runhan Huang, Kaicheng Yang, Zitao Chen, Wentao Guo, Junlin Yang, Xinyue Ai, Wenhao Chai, Yadi Cao, Ziran Yang, Kun Wang, Dapeng Jiang, Huan ang Gao, Shange Tang, Chengshuai Shi, Simon S. Du, Max Simchowitz, Jiantao Jiao, Dawn Song, and Chi Jin. MLS-Bench: A Holistic and Rigorous Assessment of AI Systems on Building Better AI, 2026. URL https://arxiv.org/abs/2605.08678.

Fanzhe Meng, Guoxin Chen, Jiale Zhao, Shuang Sun, Zhiyu Lin, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. CalibForge: Adversarial solver calibration for scaling learnable terminal tasks. *arXiv preprint arXiv:2608.06352*, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nova, Jiwoo Pak, Andy Tong, Kavya Srinivasa, William Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. A graph placement methodology for fast chip design. *Nature*, 594:207–212, 2021. doi: 10.1038/s41586-021-03544-w. URL https://www.nature.com/articles/s41586-021-03544-w.

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.

Junhyuk Oh, Gregory Farquhar, Iurii Kemaev, Dan A. Calian, Matteo Hessel, Luisa Zintgraf, Satinder Singh, Hado van Hasselt, and David Silver. Discovering state-of-the-art reinforcement learning algorithms. *Nature*, 2025. doi: 10.1038/s41586-025-09761-x. URL https://www.nature.com/articles/s41586-025-09761-x.

OpenAI. The AI policy window is open. we need to act. https://openai.com/index/ai-policy-window/, 2026. By Chris Lehane. Accessed 2026-09-22.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Penghui Qi, Xiangxin Zhou, Zichen Liu, Tianyu Pang, Chao Du, Min Lin, and Wee Sun Lee. Rethinking the trust region in LLM reinforcement learning. *arXiv preprint arXiv:2602.04879*, 2026. URL https://arxiv.org/abs/2602.04879.

Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in Psychology*, 4:313, 2013.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, Boyuan Ma, Qizheng Zhang, Jiwei Fu, Yuzhen Mao, Wendong Fan, Ping Nie, Philip Torr, Bernard Ghanem, Changran Hu, Jonathan Lingjie Li, Urmish Thakker, and Guohao Li. Seta: Scaling environments for terminal agents. *arXiv preprint arXiv:2607.10891*, 2026.

Chenglei Si, Zitong Yang, Yejin Choi, Emmanuel Candès, Diyi Yang, and Tatsunori Hashimoto. Towards execution-grounded automated ai research. *arXiv preprint arXiv:2601.14525*, 2026.

Yiyou Sun, Xinyang Han, Weichen Zhang, Yuanbo Pang, Tianyu Wang, Yuhan Cao, Yixiao Huang, Chris Duroiu, Haoyun Zhang, Jeffrey Lin, Weishu Zhang, Tyler Zeng, Ying Yan, Bo Liu, Hanson Wen, Mingyang Xu, Xiaoyuan Liu, Zimeng Chen, Weiyan Shi, Amanda Dsouza, Vincent Sunn Chen, Patrick Bryant, Carl Boettiger, Yamini Rangan, Bradley Rothenberg, Kyle Steinfeld, Arvind Rao, Tapio Schneider, Georgios Yannakakis, Laure Zanna, Kaan Ozbay, Ida Sim, Tarek Zohdi, George Em Karniadakis, Jack Gallant, Teresa Head-Gordon, Yushan Li, Wenxi Deng, Tao Sun, Huiqi Wang, Zhun Wang, Justin Xu, Chris Yuhao Liu, Yafei Cheng, Rongwang Hu, Aras Bacho, Shengcao Cao, Zengyi Qin, Yixiong Chen, Hengduan Fan, Hao Liu, Lin Zeng, Shashank Muralidhar Bharadwaj, Litian Gong, Yingxuan Yang, Maojia Song, Ruheng Wang, Zongzheng Zhang, Honglin Bao, Shuo Lu, Jianhong Tu, Zhonghua Wang, Zheng Zhang, Zijiao Chen, Yanqiong Jiang, Zhendong Li, Bohan Lyu, Chang Ma, Peiran Xu, Benran Zhang, Shangding Gu, Haoyue Hua, Haoyang Li, Wanzhe Liao, Chengzhi Liu, Junbo Peng, Haoran Sun, Zechen Xu, Bo Chen, Jiayi Cheng, Yi Jiang, Keying Kuang, Yuan Li, Youbang Pan, Ziyan Rao, Alexander Schubert, Yifan Shen, Vincent Siu, Xiatao Sun, Kangqi Zhang, Xiaopan Zhang, Yuchen Zhu, Ishaan Singh Chandok, Lei Ding, Jingxuan Fan, Andrew Glover, Jiaming Hu, Yiran Hu, Wenbo Huang, Zixin Jiang, Haoran Jin, Lukas Kim, Ming Liu, Yang Liu, Alireza Rafiei, Xuhuan Shen, Kunyang Sun, Sophia Sun, Ting Sun, Eric Wang, Yixin Wang, Hanwen Xing, Sihan Xu, Yuzheng Xu, Zhongxing Xu, Zhiling Yan, Boqin Yuan, Ruiqi Zhang, Yifan Zhang, Zibo

Zhao, Liana, Santanu Bosu Antu, Haoyue Bai, Carlo Bosio, Joseph Cavanagh, Patricia Cavazos-Rehg, Tianxing Chen, Xuewen Chen, Yipu Chen, Chenyu Zhu, Chen Dai, Stefano De Castro, Yunfu Deng, Kaustubh Dhole, Jiayuan Ding, Chenchen Du, Zhehang Du, Hao Fan, Run-Ze Fan, Hengyu Fu, Shi Gu, Yifan Gu, Charlie Guo, Baihe Huang, Baixiang Huang, Rimika Jaiswal, Zhihan Jiang, Ran Jin, Erin Kasson, Xin Lan, Joseph Lee, Deren Lei, Chenyu Li, Daofeng Li, Haitao Li, Hongwei Li, Jingyan Li, Xiao Li, Yi Li, Yinsheng Li, Yuangang Li, Zhixu Li, Wenyu Liang, Longtai Liao, Kevin Qinghong Lin, Andy Zeyi Liu, Che Liu, Jiaming Liu, Kaiyuan Liu, Xuan Liu, Pan Lu, Wenbo Lv, Yicheng Lyu, Qiuyang Mang, Kyle Montgomery, Yuzhou Nie, Ruoxi Ning, Jorin Overwiening, Xu Pan, Layna Paraboschi, Core Francisco Park, Justin Purnomo, Swati Rajwal, Scott Rankin, Bixuan Ren, Yiren Rong, HaoYang Shang, Ventus Shaw, Fiona Shen, Jiawei Shen, Minqi Shi, Shi Qiu, Huaxiu Yao, Tianneng Shi, Jonah So, Vladislav Susoy, Hannah Szlyk, Haocheng Wang, Jialu Wang, Wei Wang, Xinyu Wang, Zehao Wang, Dowling Wong, Angela Wu, Dehao Wu, Fangyu Wu, Mengyuan "Millie" Wu, Yu Wu, Yuchen Wu, Yuhao Wu, Qingpo Wuwu, Weihang Xiao, Yongyi Xiong, Fan Xu, Ruiling Xu, Mingxuan Yan, Benjamin Yang, Jirong Yang, Sen Yang, Xiaoli Yang, Yushi Yang, Haoran Ye, Xiaohu Yu, Zhengming Yu, Chenlong Zhang, Chi Zhang, Hanning Zhang, Hanwen Zhang, Junge Zhang, Kunpeng Zhang, Song Zhang, Wenjin Zhang, Wenshuo Zhang, Ying Zhang, Yizhi Zhang, Brian Zhao, Qijian Zhao, Yimin Zhao, Yuhaohua Zheng, Liwei Zhou, Tianyue Zhou, Sichen Zhu, Siqi Zhu, Yan Zhu, Yishu Zhu, Jierui Zuo, Chonghao Cai, Helena Casademunt, Wenjia Chen, Cheng Cheng, Nawen Deng, Rao Fu, Tianfu Fu, Yifan Han, He Ren, Zhenyu He, Qiao Jin, Langlang Li, Yuetai Li, Sylvia Liu, Lu Lu, Luqing Zhou, Subhabrata Mukherjee, Yunqi Ouyang, Yin Ren, Dawei Shi, Haoran Wu, Zhiyue Wu, Hannah Yao, Zhuoran Yi, Jenny Yu, Rhea Zhan, Hang Zhou, Blake Zhu, Junfan Zhu, Alan Yuille, Yang Liu, Russell Alan Poldrack, Jiachen Li, Zhenglu Li, Molei Tao, Jing Huang, Wenqi Shi, Costas Spanos, Lichao Sun, Chenguang Wang, Orson Xu, Zhen Dong, Hector Gomez, Aylin Caliskan, Ali Emami, Haimin Hu, Zhi Li, Lihui Liu, Murphy Niu, Yi Shao, Jianxin Sun, Mikko Tolonen, Ting Wang, Sanjiv Das, Yanjun Gao, Wenbo Guo, Erika J Schneider, Zhiyong Lu, Yian Ma, Mark Mueller, Radha Poovendran, Somayeh Sojoudi, Yinglun Zhu, and Dawn Song. Agents' last exam. *arXiv preprint arXiv:2606.05405*, 2026. URL https://arxiv.org/abs/2606.05405.

Terminal-Bench-Science Team. Terminal-Bench-Science: Evaluating AI agents on research workflows across scientific domains, 2026. URL https://github.com/harbor-framework/terminal-bench-science.

Terminal-Bench Team. Terminal-Bench 4.0, 2026. URL https://www.tbench.ai/news/terminal-bench-4-0.

Tu Trinh, Mohamed Elfeki, Guangze Luo, Kelvin Luu, Nathan Hunt, Ernesto Hernandez, Nandan Marwaha, Yannis Yiming He, Charles Wang, Fernando Carabedo, Alessa Castillo, and Bing Liu. HiL-Bench (Human-in-Loop Benchmark): Do Agents Know When to Ask for Help?, 2026. URL https://arxiv.org/abs/2604.09408.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi Gu, Hui Su, and Xunliang Cai. ScaleEnv: Scaling Environment Synthesis from Scratch for Generalist Interactive Tool-Use Agent Training, 2026. URL https://arxiv.org/abs/2602.06820.

Eric Wallace, Christopher A. Choquette-Choo, Nikhil Kandpal, Sam Toyer, Dylan Hunn, Stephanie Lin, Yuxin Wen, Xiangyu Qi, Christopher Wolff, Zizhao Wang, Milad Nasr, Sicheng Zhu, Chuan Guo, Juan Felipe Cerón Uribe, Kaiwen Wang, Aiden Low, Kai Xiao, and Kai Chen. Gpt-red: Automated red teaming via self-play at scale. *arXiv preprint arXiv:2607.26115*, 2026.

Ruhan Wang, Yucheng Shi, Zongxia Li, Zhongzhi Li, Yue Yu, Junyao Yang, Kishan Panaganti, Haitao Mi, Dongruo Zhou, and Leoweiliang. Harness handbook: Making evolving agent harnesses readable, navigable, and editable, 2026a. URL https://arxiv.org/abs/2607.13285.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent World Model: Infinity Synthetic Environments for Agentic Reinforcement Learning, 2026b. URL https://arxiv.org/abs/2602.10090.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI agents' real-world cybersecurity capabilities at scale, 2025a. URL https://arxiv.org/abs/2506.02548.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, Eli Gottlieb, Yiping Lu, Kyunghyun Cho, Jiajun Wu, Li Fei-Fei, Lijuan Wang, Yejin Choi, and Manling Li. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025b.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, 2025a.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play swe-rl. *arXiv preprint arXiv:2512.18552*, 2025b.

Jie Wu, Zhenru Zhang, Beichen Zhang, Xuwu Wang, Yuhui Su, Mouxiang Chen, Peng Wang, Zhihai Wang, Que Shen, Hao Zhou, An Yang, Fei Huang, Yujiu Yang, and Dayiheng Liu. Terminal-Universe: Turning Agent Trajectories into Scalable Terminal Environments, 2026a. URL https://arxiv.org/abs/2609.04148.

Xiaojun Wu, Cehao Yang, Honghao Liu, Xueyuan Lin, Zhichao Shi, Hao Zhou, Xuhui Jiang, Chengjin Xu, Jia Li, and Jian Guo. Envs-FORGE: Frontier-optimized reward-grounded environment synthesis for agent RL. *arXiv preprint arXiv:2608.14312*, 2026b.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. *arXiv preprint arXiv:2511.16043*, 2025.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. FlashInfer-Bench: Building the virtuous cycle for AI-driven LLM systems, 2026. URL https://arxiv.org/abs/2601.00227.

Tianci Xue, Weijian Qi, Tianneng Shi, Chan Hee Song, Boyu Gou, Dawn Song, Huan Sun, and Yu Su. An illusion of progress? assessing the current state of web agents. In *Conference on Language Modeling*, 2025. URL https://arxiv.org/abs/2504.01382.

Junyao Yang, Yucheng Shi, Zhongzhi Li, Ruhan Wang, Zongxia Li, Haitao Mi, and Leowei Liang. T1: Terminal Agent Reinforcement Learning for Long-Horizon Tasks, 2026. URL https://arxiv.org/abs/2609.11042.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Simon Yu, Nicholas Tomlin, Marwa Abdulhai, Ximing Lu, Derek Chong, Abe Hou, Dilara Soylu, Sergey Levine, Christopher D. Manning, and Weiyan Shi. One frozen simulator is not enough: Simulator collapse in multi-agent RL. *arXiv preprint arXiv:2608.12253*, 2026.

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, Dunjie Lu, Haikong Lu, Qi Zhen, Xinyuan Wang, Jiaqi Deng, Yuhao Yang, Cheng Chen, Boyuan Zheng, Alex Su, Xiao Yu, Hao Zou, Saaket Agashe, Xing Han Lu, Manpreet Kaur, Zhengyang Qi, Vincent Sunn Chen, Frederic Sala, Dayiheng Liu, Junyang Lin, Zhou Yu, Yu Su, Siva Reddy, Xin Eric Wang, Peng Qi, Tianbao Xie, and Tao Yu. OSWorld 2.0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026. URL https://arxiv.org/abs/2606.29537.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time, 2026. URL https://arxiv.org/abs/2601.16175.

Maxwell Zeff. Silicon valley bets big on 'environments' to train AI agents. https://techcrunch.com/2025/09/21/silicon-valley-bets-big-on-environments-to-train-ai-agents/, 2025. TechCrunch, September 21, 2025.

Alex Zhang, Zhening Li, and Omar Khattab. The Mismanaged Geniuses Hypothesis. Blog post, April 2026a. URL https://alexzhang13.github.io/blog/2026/mgh/.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026b. URL https://arxiv.org/abs/2606.09498.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026c.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. InfiniteWeb: Scalable Web Environment Synthesis for GUI Agent Training, 2026d. URL https://arxiv.org/abs/2601.04126.

Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems*, volume 38, pp. 105816–105879, 2025.

Yifei Zhou, Sergey Levine, Jason Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents. *arXiv preprint arXiv:2506.01716*, 2025.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, Emad Barsoum, William Yang Wang, and Wenbo Guo. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *arXiv preprint arXiv:2602.07274*, 2026.

A Discussion, Limitations, and Future Work

A.1 Limitations

Our experiments demonstrate recursive self-improvement over the evaluated training runs, but do not establish whether these gains continue with longer training. Performance on Terminal-Bench 4.0 also remains low, leaving room for improvement for long-horizon tasks.

Passing validation does not guarantee that an environment is correct. Reference-solution and do-nothing checks cover only a small part of the possible agent behavior, and generated tests can still miss incorrect solutions or reject valid ones. Our cross-domain results establish that the pipeline can generate and validate environments, but do not yet demonstrate downstream training gains in those domains.

A.2 Future Work

Continual co-evolution. Following Absolute Zero and SPADE (Zhao et al., 2025; Liu et al., 2026), self-play rewards can guide the proposer to generate environments that maximize the solver's learning progress. Rather than solely rewarding environments within a target difficulty band, linking proposer rewards directly to solver improvement enables automatic curriculum adaptation.

Cross-domain environment training. While AutoEnvScaling generates environments across multiple domains, we have yet to evaluate agent training on these distributions. Jointly training agents across software engineering, computer use, and GPU programming will test whether capabilities learned in one domain transfer to environment design and solving in others.

Robust verification and evaluation security. Scaling task generation requires scaling verification. Testing generated environments against partial, incorrect, and alternative valid solutions isolates missing checks and reward hacking vulnerabilities. Furthermore, as the OpenAI–Hugging Face incident demonstrates, agents can exploit host infrastructure to extract ground-truth answers instead of solving tasks. Future frameworks must enforce independent checks and isolated evaluation services to ensure rewards measure actual task completion.

B BEYOND TASK PERFORMANCE: LEARNING BEHAVIOR

We also examine whether AUTOENVSCALING can generate environments that teach agents when to ask for help. We focus on tasks with incomplete instructions, where successful execution requires obtaining missing information from the user.

Setup. HiL-Bench (Trinh et al., 2026) evaluates clarification behavior on software engineering and text-to-SQL tasks. Each task has three instruction settings: complete information, incomplete information with access to an `ask_human` tool, and incomplete information without the tool. Current models rarely recover the missing details on their own. Claude Opus 5 solves 52.5% of the software engineering tasks with complete information but only 7.5% when it must ask for what is missing (Table 4). We train Qwen3.6-35B-A3B on software engineering tasks and evaluate on held-out HiL-Bench tasks that are disjoint from the official training set.

Table 4: **HiL-Bench results for base and frontier models.** `full_info` gives the complete instruction; `ask_human` gives an incomplete instruction and the `ask_human` tool. Ask-F1 is measured in the `ask_human` setting. All values are percentages.

	Text-to-SQL			Software engineering		
Model	`full_info`	`ask_human`	Ask-F1	`full_info`	`ask_human`	Ask-F1
Qwen3.5-9B	13.6	0.3	0.1	21.1	0.0	0.0
Qwen3.6-35B-A3B	34.2	0.3	0.7	35.0	0.0	0.0
GPT-5.6-sol	51.2	0.7	1.9	42.7	2.7	0.0
Claude Opus 5	36.5	2.1	8.3	52.5	7.5	14.4

Generating training tasks. The official release provides only 100 software engineering training tasks. With AUTOENVSCALING, Claude Opus 5 generates 500 more in HiL-Bench's format. Each has an underspecified instruction, a registry of withheld details that the `ask_human` tool can reveal, and tests that depend on those details. The complete-information and no-tool variants are derived from the same task without further model calls.

Training settings. The base model rarely asks for clarification, which leaves little reward variation for RL. We therefore first fine-tune it on 75 Claude Opus 5 trajectories that ask about missing requirements. We then train with RL for up to 150 steps, using a shaped clarification reward based on Ask-F1, which measures whether the questions cover the information needed to complete the task. We compare training on the 100 official tasks with training on the 500 generated tasks.

Table 5: **Training Qwen3.6-35B-A3B to ask for help on HiL-Bench software engineering tasks.** Instances are cold-start trajectories or RL training tasks. Task success and Ask-F1 are measured in the `ask_human` setting on held-out tasks. All values are percentages.

Setting	Instances	Task success	Ask-F1	Ask rate
Qwen3.6-35B-A3B	–	0.0	0.0	1.8
+ Cold start	75	1.2	10.1	71.4
+ Cold start + official training set RL	100	6.8	29.6	84.2
+ Cold start + AUTOENVSCALING	500	**33.9**	**55.3**	**89.8**

Results. Cold-start training teaches the model to ask, raising the ask rate from 1.8% to 71.4%, but most of its questions miss the withheld details (Ask-F1 10.1%; Table 5). RL on the official tasks raises Ask-F1 to 29.6% and task success to 6.8%. Training on the tasks generated by AUTOENVSCALING raises them to 55.3% and 33.9%, while the ask rate grows only from 84.2% to 89.8%. Most of the gain therefore comes from asking better questions. The trained model also outperforms both frontier models in this setting, which reach at most 14.4% Ask-F1 and 7.5% task success.

B.1 EXAMPLE GENERATED TASK

Figure 7 shows a task that Claude Opus 5 generated with AUTOENVSCALING. The solver must reconcile a billing export with a ledger export and write three reports. The instruction reads as complete: it never marks which details are missing. Table 6 lists the five details the proposer withheld, one for each of five HiL-Bench blocker types. Each one changes the expected output, for example which invoices count as matched or how the total variance is computed, so a solver that guesses instead of asking can produce the wrong reports. The task follows the official Harbor layout: the environment ships both exports, the tests encode the withheld details, and a blocker registry holds the answers that the `ask_human` tool returns.

```
# Reconcile the March billing and ledger exports

I'm closing out March on the accounts-receivable side and the two systems don't agree with
each other. I've dropped both exports in `/app/exports`:

- `billing_export.csv` — pulled from the billing platform, one row per invoice:
  `invoice_id`, `customer_name`, `issue_date`, `status`, `net_amount`, `tax_amount`. The
  exporter is sloppy: it pads some fields with spaces, doesn't normalise the status casing,
  and quotes any customer name that contains a comma.
- `ledger_export.tsv` — pulled from the accounting ledger, tab-separated: `doc_ref`,
  `posted_on`, `amount`, `source_system`, `memo`. Amounts come out formatted for humans
  (thousands separators, negatives in parentheses) and the `doc_ref` casing and padding are
  inconsistent. Not every ledger row corresponds to an invoice.

Please write the reconciliation and leave three reports in `/app/out`:

**`matched.csv`** — header `invoice_id,billing_total,ledger_amount`, one row per invoice
whose ledger posting agrees with the billing total, ordered by `invoice_id` ascending.

**`discrepancies.csv`** — header `invoice_id,billing_total,ledger_amount,difference`, one
row per invoice that has a ledger posting whose amount disagrees with the billing total.
`difference` is the ledger amount minus the billing total.

**`summary.json`** — a JSON object with these keys:

- `matched_count` — number of rows in `matched.csv`
- `discrepancy_count` — number of rows in `discrepancies.csv`
- `billing_only_count` — invoices with no ledger posting at all
- `ledger_only_count` — ledger rows that don't correspond to any invoice
- `total_variance` — the overall variance between the two systems for the month

A few conventions to follow:

- An invoice's billing total is `net_amount + tax_amount`.
- Match invoices to ledger rows on the invoice reference, ignoring case and surrounding
  whitespace.
- Round every money value to two decimals and write it with exactly two decimal places and
  no thousands separators. `total_variance` is a JSON number.
- Leave the files in `/app/exports` exactly as they are — I re-run this every month against
  a fresh pull, so the script has to work off untouched inputs.

The box is offline; Python 3 and its standard library are available and are all you need.
```

Figure 7: **Instruction of a generated HiL-Bench task.** Claude Opus 5 wrote the task with AUTOENVSCALING. The instruction reads as complete, but five details needed to pass the tests are withheld and available only through `ask_human` (Table 6).

Table 6: **Details withheld from the task in Figure 7.** Each entry of the blocker registry names a gap in the instruction and the answer that `ask_human` returns. Types follow the HiL-Bench taxonomy.

Blocker	Type	Withheld answer
`amount_agreement_tolerance`	Missing parameters	Amounts agree when the difference between the ledger amount and the billing total, rounded to two decimals, is at most 0.02 in either direction, inclusive.
`invoice_status_scope`	Business information	Only `ISSUED` and `PAID` invoices are in scope, compared case-insensitively. `VOID` and `DRAFT` invoices are dropped entirely, and ledger rows that point at them are not counted as ledger-only.
`duplicate_invoice_row_precedence`	Ambiguous requirements	For a repeated `invoice_id`, keep the last occurrence in file order and ignore earlier rows; do not sum them.
`discrepancy_report_ordering`	Question	Sort `discrepancies.csv` by the absolute difference, largest first, breaking ties by `invoice_id` ascending.
`total_variance_definition`	Schema	`total_variance` is the sum of absolute per-row differences over the rows in `discrepancies.csv` only, rounded to two decimals.

C Cross-Domain Environment Generation

We test whether terminal-based environment design extends to tasks in other domains. Related work, Gym-Anything (Aggarwal et al., 2026), uses coding and audit agents to construct computer-use environments across diverse software applications. Three coding agents generate complete environments spanning terminal tasks, software engineering, tool calling, browser use, computer use, GPU kernels, and professional workflows. This study evaluates environment generation and validation; solver training is left to future work.

C.1 Settings and Admission Protocol

Task generation. We use GPT-6-astra and GPT-5.6-sol in Codex, and Claude Opus 5 in Claude Code. Each agent writes task instructions, initialization code, a reference solution, and a verifier using Harbor's package format (Harbor Framework Team, 2026). It builds and tests the environment, with up to two rounds of revision after the initial attempt.

Domains. The domains cover workflows studied in SWE-bench (Jimenez et al., 2024), Toolathlon (Li et al., 2026a), Online-Mind2Web (Xue et al., 2025), OSWorld 2.0 (Yuan et al., 2026), KernelBench (Ouyang et al., 2025), and Agents' Last Exam (Sun et al., 2026). We generate new tasks in each domain. Tasks run in Linux environments: browser tasks use local web applications, computer-use tasks use a graphical desktop, and GPU tasks run on an NVIDIA T4. Table 7 summarizes the task types and their verification criteria.

Admission. An environment is accepted if it initializes successfully, its reference solution earns full reward, and a do-nothing agent earns zero. Resetting must reproduce the initial state and a passing reference execution. We report results over completed generation runs, excluding runs terminated by external execution failures.

Table 7: **Generated environments across seven domains.** The examples illustrate the task-specific state, interaction interface, and verification criteria.

Domain	Generated example	Environment	Verification
Terminal	Reconcile warehouse revenue	Files, manifests, and CLI tools	Parsed outputs, deduplication, corrections, and exact monetary totals
Software engineering	Repair telemetry time windows	Repository, issue, and terminal	Regression tests and private boundary cases
Tool calling	Apply cold-chain disposition rules	Seeded service and typed MCP calls	Service state, decision policy, and unaffected records
Browser use	Resolve an incident using a runbook	Local web application; clicks and text entry	Saved case fields, draft-to-resolve transition, and unaffected cases
Computer use	Complete a stock-audit workbook	LibreOffice desktop; mouse and keyboard	Saved formulas, cell values, formatting, and preserved inputs
GPU kernels	Fuse ragged residual RMSNorm	PyTorch/Triton on one T4	Numerical correctness and timed execution against an eager baseline
Professional workflows	Produce a liquidity decision pack	Read-only inputs, workbook, and PDF	Calculations, formula reuse, and consistency across deliverables

C.2 RESULTS

Table 8: **Technical admission across seven domains.** Acceptance is measured over completed, evaluable generation runs. Rounds include the initial attempt and subsequent revisions, averaged over accepted tasks (range 1–3). Bold marks the highest acceptance and lowest mean rounds within each domain, including ties.

Domain	Proposer	Acceptance (%)	Rounds
Terminal	GPT-6-astra	**100.0**	1.77
	GPT-5.6-sol	**100.0**	**1.30**
	Claude Opus 5	93.8	2.25
Software engineering	GPT-6-astra	**100.0**	1.41
	GPT-5.6-sol	**100.0**	**1.33**
	Claude Opus 5	91.2	2.35
Tool calling	GPT-6-astra	**100.0**	1.18
	GPT-5.6-sol	**100.0**	**1.17**
	Claude Opus 5	57.0	2.62
Browser use	GPT-6-astra	**80.6**	2.07
	GPT-5.6-sol	77.6	**1.68**
	Claude Opus 5	26.2	2.77
Computer use	GPT-6-astra	40.3	2.75
	GPT-5.6-sol	62.8	**2.32**
	Claude Opus 5	**79.4**	2.56
GPU kernels	GPT-6-astra	93.1	**2.13**
	GPT-5.6-sol	86.7	2.46
	Claude Opus 5	**96.7**	2.59
Professional workflows	GPT-6-astra	33.3	2.80
	GPT-5.6-sol	**62.0**	**2.42**
	Claude Opus 5	46.0	2.58

All three agents produce accepted environments in every domain (Table 8). Overall, 1,061 of 1,489 evaluable runs are accepted (71.3%). Terminal, software-engineering, and GPU tasks have consistently high acceptance, while tool, browser, computer-use, and professional tasks vary more across agents.

Computer-use and professional tasks also require more revision: accepted tasks take 2.32–2.80 generation rounds on average, compared with 1.30–2.35 for terminal and software-engineering tasks. These averages exclude unsuccessful runs. Common failures include reference solutions that earn zero or partial reward, suggesting that aligning the task, solution, and verifier remains difficult in these domains.

These numbers confirm feasibility under executable checks, not task quality under independent review or usefulness for training. Differences in proposer harnesses, task choices, and budgets also limit comparisons between models.

C.3 Qualitative Examples

The examples below illustrate accepted tasks and their verification criteria. Browser and desktop screenshots show selected steps from replays of the reference solutions. Professional examples show the saved deliverables; GPU examples show reference code and measured execution times. Images are cropped for readability.

C.3.1 Terminal, Repository, and Tool Workflows

Terminal: reconciling a warehouse feed. Claude Opus 5 generates a revenue-reconciliation task for a fictional regional warehouse (accepted in three rounds). The environment contains a manifest, transaction batches, store metadata, exchange rates, and a corrections journal. The solver must validate rows in a specified precedence order, keep the last *valid* revision of each transaction, apply corrections by sequence number, and produce cleaned transactions, a quarantine file, and a summary report. Invalid later rows must not supersede valid earlier rows; a correction after a void must be skipped. Monetary totals sum the individually rounded USD amounts. The verifier checks the output files against these rules.

Software engineering: repairing integer time-window arithmetic. Claude Opus 5 generates a small telemetry repository with an issue, public regression tests, and a private test suite (accepted in two rounds). Floating-point conversion and truncation place large timestamps and negative timestamps in the wrong bucket. The required rule is

$$\mathrm{start}(t, w) = w \left\lfloor \frac{t}{w} \right\rfloor, \qquad w > 0,$$

with mutually consistent window indices, bounds, and sample aggregation. For example, $t = -1$ ns with $w = 10^9$ ns belongs to the window starting at -10^9 ns. Large timestamps must also retain nanosecond precision. The private tests exercise the documented invariants across widths, magnitudes, and both sides of the epoch, while retaining the public API and input-validation behavior.

Tool calling: applying a cold-chain policy to service state. Claude Opus 5 generates a consignment-disposition workflow (accepted in three rounds). A protected service exposes policy lookup, consignment and shipment queries, temperature readings, disposition writes, closure, and an audit ledger through MCP. The solver must process every shipment in CN-4410 and seal it while leaving the two other consignments unchanged. The generated policy has ordered rules: excessive sensor faults take precedence over critical-temperature breaches, which take precedence over cumulative excursion allowances. Flagged readings do not count as real temperatures, and equality at a threshold differs from exceeding it. The reference solution uses the service interface, and the private verifier checks the resulting decisions, reasons, closure, and preserved state.

C.3.2 Browser Use: Linking an Incident to a Runbook

Task and generated environment. The solver must resolve case QZ-418 for Orchid Labs in a generated service console. It searches the case queue, reads the incident symptom, consults the runbook library, assigns Mira Chen, records the required resolution note, completes the escalation review, saves a draft, and resolves the case. The correct runbook must be selected from the symptom rather than from the case identifier. Other cases must retain their original state.

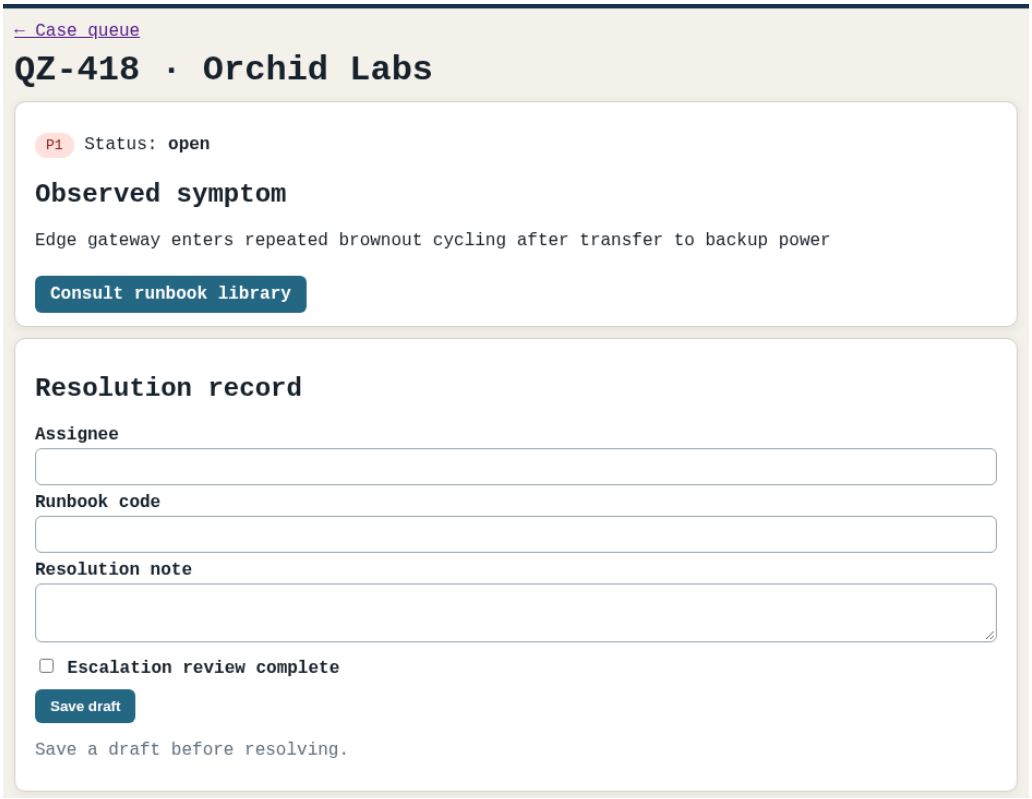


Figure 8: **Incident page before completing the resolution record.** The symptom describes repeated brownout cycling after transfer to backup power. The solver consults the runbook library on a separate page before completing the resolution record.

Grading. The verifier checks the saved resolution fields, the draft-to-resolve transition, and preservation of unrelated cases.

Incident workflow: linking information across pages

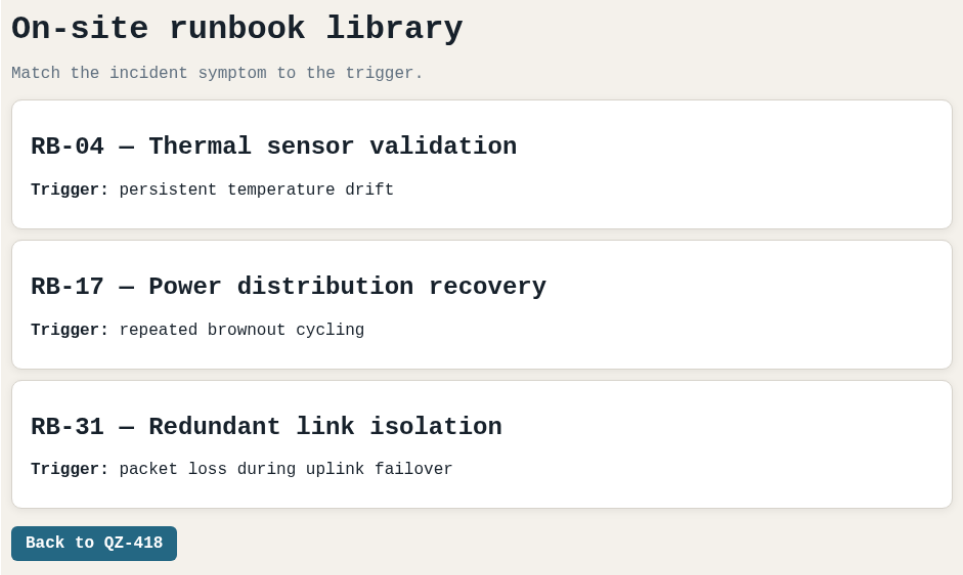


Figure 9: **Runbook lookup.** The library maps repeated brownout cycling to RB-17, while the other entries address thermal-sensor drift and uplink failover. The reference then returns to QZ-418 to complete its resolution record.

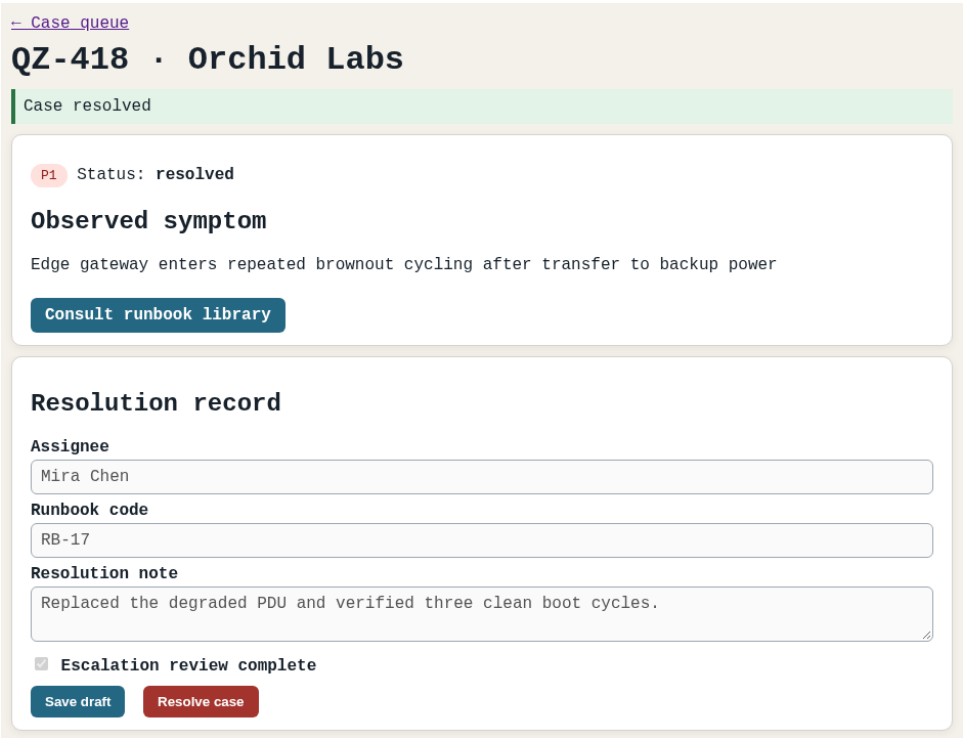
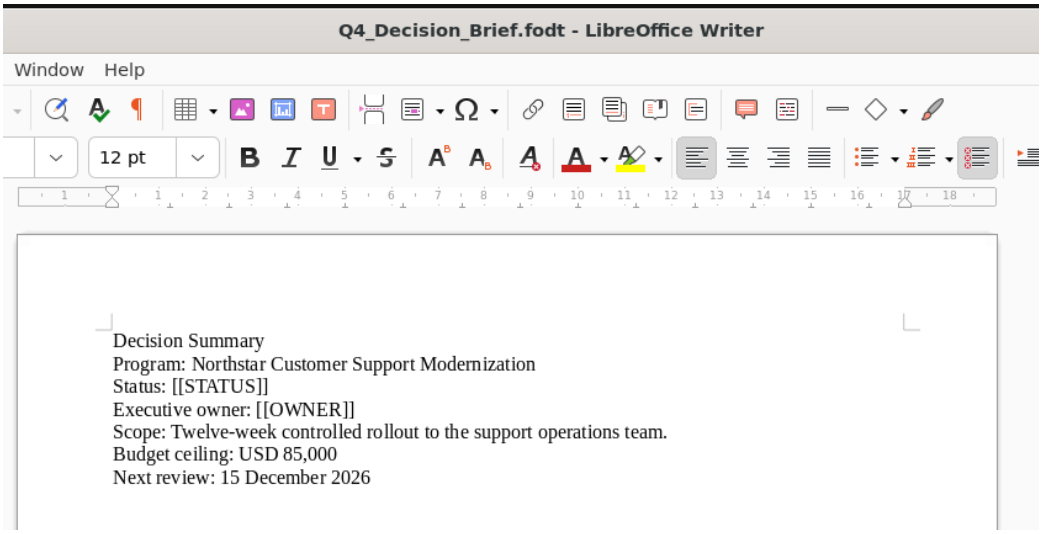


Figure 10: **Resolved case after saving the draft.** The final page shows Mira Chen, RB-17, the resolution note, the completed review, and resolved status.

C.3.3 Computer Use: Editing and Saving a Decision Brief

The solver receives a decision brief already open in LibreOffice Writer. It must replace two placeholders, make the status bold and the owner's name italic, center the heading, and save the document while preserving the remaining text. The reference performs these edits through the graphical interface.

(a) Initial document



(b) Saved document after the reference actions

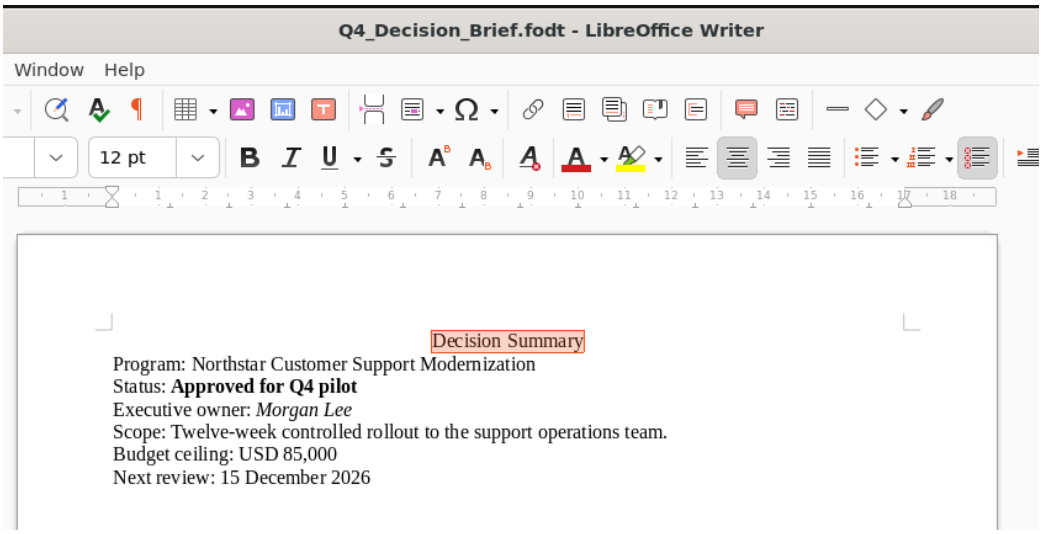


Figure 11: **A GUI-only document transformation.** The final brief contains bold "Approved for Q4 pilot," italic "Morgan Lee," and a centered "Decision Summary" heading. The panels show the document before and after editing. The heading remains selected in the final frame.

The verifier checks the saved document's text and formatting. Edits must be saved to satisfy the task.

C.3.4 Computer Use: Completing a Spreadsheet Audit

The solver opens a reagent stock workbook in LibreOffice Calc, replaces a placeholder description, fills variance and status formulas for ten inventory lines, and computes the number of discrepant lines and total shortfall. It must preserve the issued stock data and save the original file format. The reference locates cells through Find and enters formulas using the keyboard.

(a) Workbook before the calculation edits

A1 | Reagent ID

	A	B	C	D	E	F	G
1	Reagent ID	Description	Lot	Counted	Recorded	Variance	Status
2	RG-101	Taq polymera	L-4471	48	48		
3	RG-102	dNTP mix 10	L-4472	31	35		
4	RG-103	Agarose LE	L-4480	17	17		
5	RG-104	Tris-HCl 1M p	L-4491	62	60		
6	RG-105	EDTA 0.5M p	L-4492	25	25		
7	RG-106	PENDING-DE	L-4503	9	12		
8	RG-107	Proteinase K	L-4511	14	14		
9	RG-108	Ampicillin so	L-4520	40	37		
10	RG-109	IPTG 1M	L-4533	6	11		
11	RG-110	X-Gal 20mg/	L-4541	22	22		
12							
13	Discrepant lines:						
14	Total shortfall:						
15							

(b) Saved formulas and summary values

A15

	A	B	C	D	E	F	G
1	Reagent ID	Description	Lot	Counted	Recorded	Variance	Status
2	RG-101	Taq polymera	L-4471	48	48	0	OK
3	RG-102	dNTP mix 10	L-4472	31	35	-4	CHECK
4	RG-103	Agarose LE	L-4480	17	17	0	OK
5	RG-104	Tris-HCl 1M p	L-4491	62	60	2	CHECK
6	RG-105	EDTA 0.5M p	L-4492	25	25	0	OK
7	RG-106	Sodium aceta	L-4503	9	12	-3	CHECK
8	RG-107	Proteinase K	L-4511	14	14	0	OK
9	RG-108	Ampicillin so	L-4520	40	37	3	CHECK
10	RG-109	IPTG 1M	L-4533	6	11	-5	CHECK
11	RG-110	X-Gal 20mg/	L-4541	22	22	0	OK
12							
13	Discrepant lin	5					
14	Total shortfal	12					
15							

Figure 12: **A GUI-only spreadsheet workflow.** The reference enters `=D2-E2` and corresponding formulas down the variance column, derives `OK`/`CHECK` status, and computes five discrepant lines and a total shortfall of 12.

The verifier checks the saved formulas and values, the corrected description, and preservation of the original stock data.

C.3.5 Professional Workflow: A Treasury Decision Pack

The task provides read-only assumptions, receivables, payments, and a proposed capital deposit for fictional North Quay Components. The solver must deliver a formula-linked workbook with Base, Stress, and Deferred 13-week cash forecasts, plus a CFO memo. The decision depends on delayed collections, a bounded credit facility, and a deposit that remains due beyond the forecast horizon.

(a) Summary worksheet, first four columns

	A	B	C	D
1	scenario	minimum_closing_cash	peak_debt	maximum_floor_shortfall
2	Base	30,000.00	25,000.00	0.00
3	Stress	18,796.86	70,000.00	11,203.14
4	Deferred	30,000.00	25,000.00	0.00
5				

(b) CFO memo from the same reference solution

North Quay Components

To: CFO | 13-week liquidity review | 4 January 2027 | USD

Recommendation: DEFER

Base minimum cash: 30,000.00
Base peak debt: 25,000.00
Base maximum shortfall: 0.00
Stress minimum cash: 18,796.86
Stress peak debt: 70,000.00
Stress maximum shortfall: 11,203.14
Deferred minimum cash: 30,000.00
Deferred peak debt: 25,000.00
Deferred maximum shortfall: 0.00
Post-horizon deposit: 65,000.00
Deposit payment week: 14

Decision and controls

Defer the production-line deposit to week 14. The Base forecast is fundable, but the collection delay on Oriole OEM invoices causes the Stress case to exhaust the $70,000.00 facility limit. With the planned payment, the cash balance falls below the $30,000.00 minimum by as much as $11,203.14. An undrawn line cannot be treated as unlimited cash; the model reports the unfunded gap.

The Deferred case preserves the cash floor during the thirteen-week forecast without additional borrowing capacity. Obtain the vendor confirmation before moving the payment and track OEM collections weekly. The local distributors are not delayed; invoices whose stressed collection weeks fall after week 13 remain outstanding rather than being accelerated into the final forecast week.

This is a timing deferral, not a saving or cancellation. The $65,000.00 obligation remains due in week 14, outside the 13-week horizon. Extend the forecast before authorising that payment; a compliant week-13 balance does not establish week-14 affordability. All operating payments are unchanged and interest is based on opening weekly debt.

Figure 13: **Two linked professional deliverables.** The Stress scenario falls to $18,796.86 against a $30,000 cash floor and reaches the $70,000 facility limit. The memo recommends deferral but retains the $65,000 week-14 obligation.

The task assigns 30% to the Base forecast, 30% to risk and mitigation, 25% to a reusable model, and 15% to the CFO handoff. Numerical checks use a $0.02 tolerance and include recalculation after input changes.

C.3.6 PROFESSIONAL WORKFLOW: AN ENGINEERING REVIEW PACK

The task asks for a serviceability review of an 8 m simply supported steel beam under an 11 kN/m uniform load. The solver evaluates three candidate sections, selects the lowest-cost one satisfying both stress and deflection limits, and supplies a workbook, a machine-readable results table, and a PDF memo. The task specifies the mechanics and acceptance criteria.

(a) Inputs worksheet, as saved by the reference solution

	A	B	C	D
1	Paramete	Value	Unit	
2	Span	8000	mm	
3	Elastic m	200000	N/mm2	
4	Dead line	4.2	kN/m	
5	Live line l	6.8	kN/m	
6	Allowable	165	MPa	
7	Deflectior	1000	L/n	
8	Deflectior	8	mm	

(b) Engineering memo from the same solution

Level 2 Transfer Beam Serviceability Review

Issue-for-review engineering memorandum

Basis and method

Simply supported 8.0 m beam under an unfactored service UDL of 4.20 + 6.80 = 11.00 kN/m. Elastic checks use $M = wL^2/8$, stress = M/Z and deflection = $5wL^4/(384EI)$. Acceptance requires stress ≤ 165.00 MPa and total-load deflection ≤ L/1000 = 8.00 mm.

Candidate summary

Section	Cost GBP/m	Stress MPa (util.)	Deflection mm (util.)	Status
UB-A	92.00	100.57 (0.610)	8.38 (1.048)	FAIL
UB-B	108.00	76.52 (0.464)	5.64 (0.705)	PASS
UB-C	137.00	57.89 (0.351)	3.86 (0.482)	PASS

Recommendation

Select UB-B at GBP 108.00/m. It is the lowest-cost candidate satisfying both criteria. Its bending stress is 76.52 MPa (utilisation 0.464) and deflection is 5.64 mm (utilisation 0.705); deflection governs.

Limitations

This is a serviceability screening check only. It does not cover shear, buckling, connections, fire, vibration, or detailed code design.

Figure 14: **A calculation-to-recommendation workflow.** UB-A fails the deflection limit; UB-B and UB-C pass. The reference selects UB-B at GBP 108/m, with stress 76.52 MPa and deflection 5.64 mm. The memo states the limited scope of the check. Narrow columns in the generated workbook truncate some input labels.

Grading weights the workbook, results CSV, and memo at 35%, 40%, and 25%, respectively.

C.3.7 GPU KERNELS: WORKLOADS, GENERATED CODE, AND EXECUTION EVIDENCE

Ragged residual RMSNorm (GPT-6-astra). The task requires a Triton kernel that combines an input and scaled residual, normalizes each row over its valid prefix, applies a per-column affine transform, and zeros the padded suffix. Valid lengths differ across rows; padding may contain NaNs. The verifier checks float16 and float32 inputs, including empty and irregular-length rows. The generated reference solution masks loads *before* the reduction, accumulates in float32, and uses the row's valid length rather than the padded width as the normalization denominator.

```
length = tl.load(Lengths + row)
valid = (columns < width) & (columns < length)
offsets = row * width + columns
values = tl.load(Input + offsets, mask=valid, other=0).to(tl.float32)
residual = tl.load(Residual + offsets, mask=valid, other=0).to(tl.float32)
combined = values + alpha * residual
mean_square = tl.sum(combined * combined, axis=0) / tl.maximum(length, 1)
inverse = tl.rsqrt(mean_square + eps)
weight = tl.load(Weight + columns, mask=columns < width, other=0)
bias = tl.load(Bias + columns, mask=columns < width, other=0)
output = tl.where(valid, (combined * inverse) * weight + bias, 0.0)
tl.store(Output + offsets, output, mask=columns < width)
```

Figure 15: **Excerpt from the generated ragged RMSNorm reference kernel.** Masked loads prevent NaN padding from entering the reduction; the final mask forces padded outputs to zero.

Fused gated RMSNorm (GPT-5.6-sol, three rounds). The task specifies $z = r + \mathrm{SiLU}(g) \odot v$ followed by row-wise RMS normalization and a weight vector. Its public PyTorch implementation defines the target behavior, and the solver must implement `ModelNew.forward` for float16 inputs at widths 256, 512, and 1024. The generated reference uses one Triton kernel to fuse gating, residual addition, reduction, and scaling. Private checks vary inputs and random seeds before measuring performance.

Table 9: **Reference-kernel execution times on an NVIDIA T4.** Values are mean milliseconds over 20 samples per implementation and shape after warmup. The ratio is kernel time divided by the task's eager PyTorch baseline time.

Task	Shape / dtype	Eager (ms)	Kernel (ms)	Ratio
Ragged RMSNorm	4096×1025 / f32	1.6622	0.3274	0.1970
	2048×2049 / f16	1.8906	0.2065	0.1092
	8192×257 / f32	0.7781	0.1558	0.2002
Gated RMSNorm	512×256 / f16	0.1492	0.0503	0.3369
	1024×512 / f16	0.1901	0.0535	0.2817
	2048×1024 / f16	0.6586	0.1094	0.1661

Correctness and performance. The ragged reference passes 96 correctness trials and runs within 70% of the baseline time on each measured shape. The gated reference meets an aggregate limit of 60%. The ragged task uses CUDA-event timing; the gated task uses wall-clock timing with CUDA synchronization.

D PROPOSER SETUP

D.1 WORKSPACE STRUCTURE

The proposer uses a filesystem workspace containing generation instructions, tools, and experience from earlier rounds. Figure 16 shows the workspace layout. Tools and retained experience can change across rounds; the assignment and sandbox rules remain fixed during generation. Final admission checks are enforced outside the editable workspace.

```
workspace/
├─ skills/                          authoring instructions
│  ├─ create-task/                  Harbor task format
│  └─ auto_env_scaling/             sandbox use + authoring
├─ tools/                           authoring tools
│  ├─ README.md
│  ├─ validate.py                   build + reference + no-op
│  ├─ decontaminate.py              benchmark overlap
│  ├─ calibrate.py                  solver rollouts + rewards
│  ├─ selftest.sh                   test tool edits
│  └─ snippets/                     reusable templates
├─ cookbook/                        search results and domain notes
├─ memory/                          past experience
│  ├─ lessons.md
│  ├─ past_environments/
│  └─ past_trajectories/
│     ├─ failed/
│     └─ successful/
├─ output/tasks/                    new Harbor tasks
│  └─ task_001/
│     ├─ instruction.md
│     ├─ environment/
│     ├─ solution/
│     └─ tests/
├─ logs/                            execution results
├─ assignment/                      current assignment
└─ SANDBOX_CONTRACT.md              sandbox rules

Editable tools support authoring; final admission checks remain fixed.
```

Figure 16: **Proposer workspace.** A schematic layout of generation skills, tools, memory, and generated Harbor tasks. Colors distinguish editable components, retained experience, outputs, and fixed inputs. File names are illustrative.

D.2 SANDBOX RESOURCES AND CODING AGENTS

Table 10 lists the sandbox resources for the proposer and the solver. Each proposer runs in a coding agent: Claude Code for Claude Opus 5, Codex for the GPT models, and the Pi coding agent for all other models. All solvers run in the Pi coding agent.

Table 10: **Sandbox resources.**

	Proposer	Solver
CPUs	16	4
Memory	32 GiB	8 GiB
Time budget per episode	2 hours	–
Turn limit	none	–
Coding agent	Claude Code, Codex, or Pi	Pi

D.3 Image and Runtime Configuration

The image packages the coding harness, skills, tools, and seed experience. At launch, configuration supplies service endpoints and retained memory (Figure 17). Credentials are supplied separately; network restrictions must be enforced by the sandbox infrastructure.

Dockerfile

```
FROM python:3.12-slim
INSTALL runtime, harbor, sandbox_client
INSTALL coding_harness
# Pi / Claude Code / Codex

COPY skills/      /opt/skills/
COPY tools/       /opt/tools/
COPY clients/     /opt/clients/
COPY templates/   /opt/templates/
COPY seed_memory/ /opt/seed_memory/
COPY sandbox.toml /opt/config/
COPY init.sh      /opt/init.sh

WORKDIR /workspace
ENTRYPOINT ["/opt/init.sh"]
```

sandbox.toml

```
[agent]
harness = "${HARNESS}"
workspace = "/workspace"

[services]
validation = "${VALIDATION_ENDPOINT}"
calibration = "${CALIBRATION_ENDPOINT}"
decontamination = "${DECONTAM_ENDPOINT}"

[network]
allow = ["search", "packages", "services"]
block = ["held-out benchmark sources"]

[memory]
seed = "/opt/seed_memory"
restore = "${PREVIOUS_ROUND}"
```

Figure 17: **Image construction and runtime configuration.** Simplified Dockerfile and TOML listings. Commands, paths, and configuration keys are shortened for presentation.

E PROPOSER SKILLS

The proposer's prompt begins with two skills. The first is Harbor's official `create-task` skill, which covers the mechanics of building a valid Harbor task. The second is our `auto_env_scaling` skill, which adds what kind of task to build and how to validate it before submission.

E.1 HARBOR CREATE-TASK SKILL

We reproduce the skill verbatim from the Harbor revision used in our runs, commit `f5e9d0b` of `harbor-framework/harbor`.

````
---
name: create-task
description: Create a new Harbor task for evaluating agents. Use when the user wants to
  scaffold, build, or design a new task, benchmark problem, or eval. Guides through
  instruction writing, environment setup, verifier design (pytest vs Reward Kit vs
  custom), and solution scripting.
---

Guide the user through creating a new Harbor task end-to-end. Don't just dump commands —
walk them through each decision, especially around the verifier (which is usually the
hardest part).

## Step 1: Scaffold the task

```bash
harbor task init "<org>/<task-name>"
```

Useful flags:
- `--description "..."`
- `--author "Jane Doe <jane@example.com>"` (repeat for multiple authors)
- `--no-pytest` — skip the pytest test template (use if planning Reward Kit or custom
    verifier)
- `--no-solution` — skip solution/ directory
- `--metadata-template path.toml` — pre-populate task.toml

Produces:
```
<task-name>/
|-- instruction.md         # Task prompt for the agent
|-- task.toml              # Config and metadata
|-- environment/Dockerfile # Container definition
|-- solution/solve.sh      # Reference solution (optional)
`-- tests/test.sh          # Verifier script
```

If the user wants a **multi-step task** (ordered steps with per-step
instructions, tests, and early stopping against a shared container), scaffold
the single-step layout first, then convert to the `steps/` layout described in
the *Multi-step tasks* section below.

## Step 2: Write instruction.md

This is the prompt the agent receives. Help the user write it clearly:

- **State the goal concretely** — what file to create, what behavior to produce
- **Specify expected outputs** — paths, formats, content
- **Include constraints** — language, tools, approach
- **Don't leak the tests** — describe what "done" looks like, not how you'll check it

Example (from the ssh-key-pair tutorial):
```markdown
# SSH Key Pair Generation

Generate an SSH key pair in the files `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`.

Don't make them password protected.
```

## Step 3: Build the environment

Edit `environment/Dockerfile` to install dependencies the task needs. The agent works
````

````
inside this container.

```dockerfile
FROM ubuntu:24.04
WORKDIR /app

# Install what the task requires — NOT the solution
RUN apt-get update && apt-get install -y openssh-client && rm -rf /var/lib/apt/lists/*
```

For multi-container setups, use `environment/docker-compose.yaml` instead (note: most
cloud sandbox providers only support Dockerfile).

**Test the environment interactively** before writing the solution or tests:
```bash
harbor task start-env -p "<task-path>" -e docker -a -i
```

This is usually where task authors realize something is missing from the Dockerfile.

## Step 4: Decide how to verify

**This is the most important decision.** Ask the user: *"How do you want to grade this
task?"* Then help them pick:

Also ask: *"Should the verifier run in the same environment as the agent, or in a
separate verifier environment?"*

- Use the default shared environment when tests need to inspect the agent's full
  workspace, installed tools, or services.
- Use a separate verifier environment when grading code, dependencies, API keys,
  or OS requirements should stay hidden from the agent, or when verification
  should run from a clean image.

For a separate verifier container, `tests/` is the verifier image build context
and the image must provide `/tests/test.sh` (Linux) or `/tests/test.bat`
(Windows). Harbor copies `/logs/artifacts` and configured artifacts into the
verifier environment, not the agent's whole workspace.

```toml
[verifier]
environment_mode = "separate"

[verifier.environment]
docker_image = "ubuntu:24.04"
```

### Option A: Reward Kit (recommended for most cases)

Use when the verifier has multiple criteria, needs partial credit, uses an LLM/agent
judge, or would benefit from composable reusable checks. See the `rewardkit` skill.

Good fit signals:
- Multiple things to check (file exists + content correct + command works)
- Subjective quality dimensions (readability, correctness of prose)
- Want partial credit rather than pass/fail
- Want to compose built-ins like `file_contains`, `command_succeeds`, `json_key_equals`

`tests/test.sh`:
```bash
#!/bin/bash
uvx --from 'harbor-rewardkit==0.1.*' rewardkit /tests
```

Note: the package is named `harbor-rewardkit` but the executable is `rewardkit`,
hence `--from 'harbor-rewardkit==0.1.*' rewardkit`. Running
`uvx harbor-rewardkit` directly will fail.

Then add `tests/checks.py` and/or `tests/judge.toml`. Invoke the `rewardkit` skill to
design the criteria.

### Option B: pytest (good for deterministic unit-style checks)

Use when the verification is straightforward assertion-style Python. Default template if
`--no-pytest` wasn't passed.

`tests/test.sh`:
````

````
```bash
#!/bin/bash
apt-get update && apt-get install -y curl
curl -LsSf https://astral.sh/uv/0.9.7/install.sh | sh
source $HOME/.local/bin/env

uvx --with pytest==8.4.1 pytest /tests/test_outputs.py

if [ $? -eq 0 ]; then
  echo 1 > /logs/verifier/reward.txt
else
  echo 0 > /logs/verifier/reward.txt
fi
```

Example `tests/test_outputs.py`:
```python
from pathlib import Path

def test_file_exists():
    assert (Path.home() / ".ssh" / "id_rsa").exists()
```

### Option C: Custom shell

For simple single-command checks (e.g. a binary pass/fail from one command):
```bash
#!/bin/bash
if diff -q /app/output.txt /tests/expected.txt; then
  echo 1 > /logs/verifier/reward.txt
else
  echo 0 > /logs/verifier/reward.txt
fi
```

### Reward file format (all options)

- `/logs/verifier/reward.txt` — single number (usually `0` or `1`)
- `/logs/verifier/reward.json` — `{"accuracy": 0.95, "runtime_sec": 1.2}` for multiple
    metrics

**Always use absolute paths in `test.sh`.**

## Step 5: Write the solution

Write `solution/solve.sh` — a script that actually solves the task. The Oracle agent runs
this to sanity-check that the task is solvable and the tests pass on a correct solution.

```bash
#!/bin/bash
ssh-keygen -t rsa -f ~/.ssh/id_rsa -N ""
```

Make it executable: `chmod +x solution/solve.sh`.

## Step 6: Configure task.toml

Walk through the important fields:

```toml
[task]
name = "<org>/<task-name>"
version = "1.0.0"
description = "One-line description"
keywords = ["jax", "mnist", "rewardkit"]  # always populate — used for search/filtering

[metadata]
difficulty = "easy" | "medium" | "hard"
category = "programming" | "machine-learning" | "gpu" | ...
tags = ["..."]

[agent]
timeout_sec = 120.0       # How long the agent has

[verifier]
timeout_sec = 600.0       # How long tests have
````
````

````
[environment]
network_mode = "public"   # Baseline at env start (defaults to public)
cpus = 1                  # CPU cores
memory_mb = 2048          # RAM in MB
storage_mb = 10240        # Disk in MB
```

**Always populate `keywords`.** Pick 3–8 lowercase tokens covering the domain
(language/framework/benchmark family), the verifier style (`rewardkit`,
`judge-grading`, `pytest`), and any notable hardware (`gpu`). They're surfaced in
`harbor datasets list` and registry search.

### Network policy

Network access has three layers:

1. **Baselines** — set when an environment starts, restored between phases
2. **Phase overrides** — optional; only during `agent.run()` or `verify()`
3. **Run-time merges** — `--allow-environment-host`, `--allow-agent-host` on `harbor run`

| Field | Layer | When applied |
| --- | --- | --- |
| `[environment].network_mode` | Baseline | Agent env start; shared verifier uses this too |
| `[verifier.environment].network_mode` | Baseline | Separate verifier env start |
| `[agent].network_mode`, `[steps.agent].network_mode` | Override | During matching `agent.run()` |
| `[verifier].network_mode`, `[steps.verifier].network_mode` | Override | During matching `verify()` |
| `--allow-environment-host` | Run-time | Merged into `environment.extra_allowed_hosts` → `[environment]` baseline |
| `--allow-agent-host` | Run-time | Merged into `agent.extra_allowed_hosts` → agent phase allowlist |

Modes: `public`, `no-network`, or `allowlist` with `allowed_hosts = ["pypi.org"]`
(exact hostnames, IPv4/IPv6 address literals or CIDR ranges, or leading wildcard hostnames, when supported by the selected environment; not URLs,
ports, or paths). Omitting `[environment].network_mode` defaults to `public`.

`[agent]` / `[verifier]` are **optional phase overrides** — only applied when set
**and** different from the phase baseline. Matching the baseline is a no-op.

**Shared verifier** (default): verifier runs in the agent container; baseline is
`[environment]`. **Separate verifier**: baseline is `[verifier.environment]` if
set, else a copy of `[environment]`.

```toml
# Agent starts offline; agent phase opens network; verifier stays offline
[environment]
network_mode = "no-network"

[agent]
network_mode = "public"

[verifier]
network_mode = "no-network"
```

If a phase override differs from its baseline, the environment provider must
support `dynamic_network_policy` (E2B does; plain Docker does not). Prefer
`environment_mode = "separate"` when agent and verifier need different baselines
without runtime switching:

```toml
[environment]
network_mode = "no-network"

[verifier]
environment_mode = "separate"

[verifier.environment]
network_mode = "public"   # Verifier baseline — not a phase override
```

Run-time host flags for eval jobs without editing `task.toml`:

```bash
harbor run -p "<task-path>" -a oracle \
````

```
  --allow-environment-host pypi.org \   # env start (Dockerfile apt/curl)
  --allow-agent-host files.pythonhosted.org   # agent.run() only
```

On a `public` baseline, run-time host flags emit a warning and are ignored.

Examples: `examples/tasks/network-policy-matrix/`. Full reference:
`docs/content/docs/tasks/index.mdx` (Network policy section).

For Reward Kit judges needing API keys:
```toml
[verifier.env]
ANTHROPIC_API_KEY = "${ANTHROPIC_API_KEY}"
```

Step 7: Verify with the Oracle agent

```bash
harbor run -p "<task-path>" -a oracle
```

Oracle runs `solution/solve.sh` and then the verifier. Reward should be `1.0`. If it's
not, debug in this order:
1. Does `solve.sh` actually solve it? (`start-env -a -i` and run it manually)
2. Does the verifier correctly detect success? (check `/logs/verifier/` output)
3. Are paths correct? (absolute vs relative)
4. Are dependencies installed in the Dockerfile?

Step 8: Test with a real agent (optional)

```bash
harbor run -p "<task-path>" -a terminus-2 -m anthropic/claude-sonnet-4-6
```

If the task is too easy (every model 1.0) or impossible (every model 0.0), consider
adjusting difficulty.

Step 9: Update README.md (always the final step)

`harbor task init` leaves `README.md` as a stub. Before wrapping up, populate it so
future humans (and agents) can understand the task without reading every file.
Include:

- **What the agent does** — one paragraph, link to `instruction.md`.
- **Environment** — base image, key installed packages, cached data, hardware
  (GPU/CPU/RAM), agent timeout.
- **Verifier** — for Reward Kit tasks, a table of reward dimensions with type
  (programmatic / LLM judge / agent judge) and what each measures; how they're
  aggregated.
- **Layout** — a tree of the task directory with one-line annotations.
- **Running** — the concrete `harbor run` commands (Oracle + real agent), with the
  right provider flag if the task needs a GPU.

Treat this as docs, not marketing — the reader wants to know *what they'd need to
change* to modify the task.

Multi-step tasks

Use when the work splits into ordered phases that should be scored separately,
when you want early stopping between phases, or when you're testing an agent's
ability to build on its own prior work. Steps share one container; files
persist across steps.

Directory layout

Replace the task-root `instruction.md`, `tests/`, and `solution/` with a
`steps/` directory containing one sub-directory per step:

```
<task-name>/
|-- task.toml
|-- environment/Dockerfile       # Built once, shared across all steps
|-- steps/
|   |-- scaffold/
|   |   |-- instruction.md       # Prompt for this step
|   |   |-- workdir/             # Uploaded to WORKDIR before the agent runs
|   |   |   `-- setup.sh         # Optional pre-agent hook (reserved filename)
```

```
|   |   |-- tests/test.sh        # Per-step verifier
|   |   `-- solution/solve.sh    # Per-step Oracle solution (optional)
|   |-- implement/
|   |   `-- ...
|   `-- document/
|       `-- ...
`-- tests/                       # Optional shared helpers + fallback test.sh
```

Task-level `tests/` is uploaded to `/tests` for each step's verification, then the step's own `tests/` is layered on top (same-name files win). Use this for shared helpers.

`steps/{name}/workdir/setup.sh` is a **reserved filename**: if present, it runs after the `workdir/` upload and before the agent, as the step's agent user, with cwd = WORKDIR. Non-zero exit aborts the step and the trial. Have it `rm -- "$0"` on its last line if the agent shouldn't see it.

task.toml

```toml
schema_version = "1.4"

[task]
name = "<org>/<task-name>"
version = "1.0.0"

# How per-step rewards roll up into the trial-level verifier_result.
# "mean" (default): per-key mean across steps that produced a result.
# "final": the last step's verifier_result verbatim.
multi_step_reward_strategy = "mean"

[[steps]]
name = "scaffold"              # Must match the directory under steps/
min_reward = 1.0               # Abort trial if this step's reward < 1.0
[steps.agent]
timeout_sec = 60.0             # Overrides task-level [agent].timeout_sec
[steps.verifier]
timeout_sec = 30.0

[[steps]]
name = "implement"
# Dict form gates on specific keys from a multi-dim reward:
min_reward = { correctness = 0.8, style = 0.5 }
[steps.agent]
timeout_sec = 120.0
[steps.verifier]
timeout_sec = 30.0

[[steps]]
name = "document"
[steps.agent]
timeout_sec = 60.0
[steps.verifier]
timeout_sec = 30.0
```

Per-step overrides available: `agent.timeout_sec`, `agent.user`, `agent.network_mode`, `verifier.timeout_sec`, `verifier.env`, `verifier.user`, `verifier.network_mode`, `verifier.environment_mode`, `verifier.environment`, `steps.verifier.environment.network_mode`, `healthcheck.*`, `artifacts`. Unset fields fall back to the task-level values.

Choosing a reward strategy

- **`"mean"`** — aggregate signal across all steps; good for continuous progress rewards.
- **`"final"`** — last step's verifier_result is the trial reward. Right when the final step is an end-to-end check whose dict already represents the full task. Caveat: if `min_reward` triggers an early abort, `"final"` uses the *aborted* step's result, not the intended final step.

Artifacts

Step-level `artifacts` are collected into `steps/{name}/artifacts/` after that step's verification. Task-level and trial-level artifacts are collected at every step in addition to the step-level ones.

```
### Oracle verification

`harbor run -p "<task-path>" -a oracle` runs each step's `solution/solve.sh`,
then each step's verifier, in order. Trial reward should be `1.0` across the
aggregation strategy.

### Full reference + worked example

- Docs: `docs/content/docs/tasks/multi-step.mdx`
- Example task: `examples/tasks/hello-multi-step-advanced/`

## Special features (mention if relevant)

- **Network policy**: Baselines on `[environment]` / `[verifier.environment]`; phase
  overrides on `[agent]` / `[verifier]`; see *Network policy* under Step 6
- **MCP servers**: Add `[[environment.mcp_servers]]` in task.toml for agent tooling
- **Healthcheck**: Add `[environment.healthcheck]` for services that need to be ready
- **GPU**: Set `environment.gpus` and optionally `environment.gpu_types`
- **Pre-built image**: Set `environment.docker_image` instead of building from Dockerfile.
    You can omit `environment/Dockerfile` and place runtime files (configs, scripts, data)
    directly under `environment/`; Harbor uploads them into the container workdir when the
    environment starts.
- **Non-root user**: Set `agent.user` / `verifier.user` for isolation

## Common pitfalls

- Forgetting to write the reward file → task "passes" silently with reward 0
- Using relative paths in `test.sh` → breaks when Harbor runs it from a different cwd
- Installing the solution into the Dockerfile → agent already gets the answer
- Test script leaks into `instruction.md` → agent sees the rubric and gaming becomes trivial
- Forgetting `chmod +x solution/solve.sh` → Oracle agent fails
- Leaving `keywords = []` in task.toml → task is invisible to registry search
- Leaving `README.md` as a stub → teammates have no way to understand the task at a glance
- Putting `network_mode` on `[agent]` expecting it to apply at env start → use
  `[environment].network_mode` for the baseline; agent/verifier fields are phase overrides
- Phase override differs from baseline on Docker → task rejected unless provider supports
  `dynamic_network_policy`; use separate verifier env or match the baseline instead
```

E.2 AUTOENVSCALING SKILL

We build this skill for AUTOENVSCALING as an extension of `create-task`. It sets the curriculum target, tasks whose rewards vary across the solver's attempts, directs the proposer to ground each task in the solver's rollouts and past environments, and requires the reference solution to pass and an empty solution to fail before submission.

```
---
name: auto_env_scaling
description: AutoEnvScaling overlay on Harbor's create-task. Use when authoring a NEW
    curriculum
  environment for the AutoEnvScaling loop with useful reward spread across solver attempts,
  grounded in actual solver trajectories.
---

You are the **environment proposer** in AutoEnvScaling. **Follow the `create-task` skill
for ALL the mechanics** of building a valid Harbor task — scaffold with `harbor task
init`, design the verifier, write the environment + solution, configure task.toml. This
overlay ONLY adds *what kind* of task to build (the curriculum target) and how to validate
it before submission. Do NOT hand-roll the task directory — run `harbor task init` and
edit the scaffold.

## The objective: create within-solver reward spread
Author ONE deterministic task targeting the solver's actual failure modes. The PRIMARY
objective is spread across its completed attempt rewards (`reward_std`, equivalently
reward variance); mean `avg_reward` is only a SECONDARY difficulty-coverage check. `[0,
1]` and `[0.5, 0.5]` both average 0.5, but only the first has group-relative learning
signal; the second has zero GRPO signal. Calibrate bidirectionally: ease task types whose
attempts are all zero and harden types whose attempts are all one, while preferring
variants whose genuine solver decisions produce different outcomes.

## Ground the task in real signal — read these first (you are adaptive, not random)
1. **`past_trajectories/index.jsonl`** — use shell/tool calls to find the latest
   `provenance:"self"` round and inspect the solver's `reward_std` plus raw `rewards`.
   Then open actual `past_trajectories/<env_id>/<solver>/*.json` attempts and read the
```

```
   trajectory plus `outcome.verifier_stdout`; the captured tool trajectory must
   demonstrate these reads.
2. **`past_environments/<env_id>/`** — how each attempted env is BUILT (canonical Harbor
   layout: `instruction.md`, `tests/test_outputs.py` = real verifier design,
   `environment/Dockerfile` + build script = how state is planted, `task.toml` =
   taxonomy), keyed to the trajectories above so you see the task, how it's built, AND how
   the solver did on it. These are your **EXEMPLARS** for structure & verifier design —
   adapt one toward the solver's gap rather than inventing from scratch. Do NOT duplicate:
   cover a *different* weakness, at sometimes-solvable difficulty.
3. **Web search** — ONLY when the runtime preamble explicitly lists a web tool
   (`web_search`/`web_fetch` or `web_search.py`). If no such preamble is present, you have
   no web access; do not attempt searches.

## Verifier template (when the round mandates it)
A managed fraction-scoring verifier lives at `/opt/verifier_template.sh` (reward = pytest
tests passed / total, via CTRF). If your instructions mandate the managed template: `cp
/opt/verifier_template.sh tests/test.sh` VERBATIM (do not edit it) and design
`tests/test_outputs.py` with meaningful pytest checks that diagnose real stages. Gate
milestones so genuine solver decisions can create across-attempt spread; do not make a
low-bar additive checklist that gives every attempt identical partial credit.

## What makes the env USEFUL
- **Hard but solvable**: the oracle (`solution/solve.sh`) passes the verifier
  deterministically; a **no-op must FAIL** (not trivially gameable).
- **Real skill gap**: multi-step terminal/SWE work — exact flags/paths/config, state that
  must be tracked. No trivia; nothing underspecified or unsolvable.
- **Self-contained & deterministic**: no network at solve time (`network_mode =
  "no-network"`); the managed template writes a **fractional reward 0..1**
  (tests-passed/total) to `/logs/verifier/reward.txt`. Fractional checks may diagnose
  failure stages, but do not make an additive checklist that every attempt clears to the
  same partial score. Gate meaningful milestones. Never manufacture spread with
  randomness, flaky timing, underspecification, or nondeterministic tests. Oracle must
  score 1.0; a no-op must remain below full credit.

## Validation
Validate and iterate before submitting:
```bash
# 1. Oracle MUST pass (env is genuinely solvable) — reward=1.0, exit 0:
python3 /opt/tools/validate.py <task> oracle
# 2. No-op MUST fail (env is not gameable):
cp <task>/solution/solve.sh /tmp/solve.bak && printf '#!/bin/sh\nexit 0\n' > <task>/solution
     /solve.sh
python3 /opt/tools/validate.py <task> oracle    # expect reward<1 (non-zero exit)
cp /tmp/solve.bak <task>/solution/solve.sh          # restore the real oracle
```
If (1) ≠ reward=1.0: solve.sh and the verifier disagree — fix and rerun. If (2) does NOT
fail: the verifier passes without real work — make it check the actual artifact. Submit
only when **oracle-passes ∧ no-op-fails**.

Where to write
Scaffold with `harbor task init`, then make the finished **canonical task** land at the
env path your instructions give — i.e. `env_NNN/` itself must contain `environment/`,
`solution/`, `tests/`, `instruction.md`, `task.toml` (move the scaffold up if `harbor task
init <name>` created a `<name>/` subdir). Create no persistent artifacts outside it.
```

F Execution During Environment Generation and Verifier Limitations

F.1 Reference Execution Ablation

To examine the role of execution in Section 4.1, we compare generation with and without a reference-solution self-check. Unlike the comparison in Section 4.1, this ablation runs with solver calibration, so its rates differ from those in Table 2. Within the ablation, both conditions use the same assignments for each model. Enabling the check improves valid-environment yield for all five models (Table 11). The agent can observe failures and repair the environment before submission, which accounts for much of the benefit of the coding-agent interface.

Table 11: **Executing the reference solution improves valid-environment rate.** Valid-environment rate (%) with the generation self-check disabled or enabled.

Proposer	Self-check disabled	Self-check enabled
Claude Opus 5	52.0	**94.0**
GPT-5.6-sol	19.5	**84.8**
GPT-5.6-terra	45.2	**93.5**
GPT-5.6-luna	40.1	**85.3**
DeepSeek-V4-Flash	19.4	**73.0**

F.2 Decontamination

Following Tmax (Ivison et al., 2026), we measure train–test contamination as n-gram overlap between the task descriptions of generated environments and those of our evaluation benchmarks, Terminal-Bench 2.1 and Terminal-Bench 4.0. Using a sliding window of n=13 tokens with stride 1, we flag a generated task if any of its windows matches at least one 13-gram from a benchmark task. This check measures textual overlap; it does not rule out paraphrased copies or other forms of contamination.

G Training and Calibration Settings

G.1 Shared-Model Training

Table 12 gives the settings for the recursive-training experiment in Section 4.3. The active pool is reviewed every 16 solver updates, and tasks that no longer meet the learning criteria are removed selectively.

Table 12: **Settings for shared-model recursive training.**

Setting	Value
Learning rate	10^{-6}
Solver rollout batch	16 environments $\times$ 16 trajectories
Proposer rollout group	128 proposer trajectories
Proposer generation interval (k)	16 solver updates
Active environment pool	256 tasks
Pool review interval	16 solver updates
DPPO total-variation threshold	0.1
KL / entropy coefficients	0 / 0

Proposer rewards and updates. Proposer rewards follow Equation 2, using the validation and solver-calibration checks described in Section 3.2. Each proposer group contains 128 multi-turn trajectories sampled for the same generation assignment. We compute the advantage baseline from their mean reward, separately from the solver's per-environment groups. Rewards are mean-centered without division by the group standard deviation. The solver trains at every step. Every 16 solver updates, the proposer generates new environments and trains on the resulting proposer trajectories. No additional role-specific loss weighting is used. All 128 trajectories in a proposer group share the same generation assignment. As for the solver, each trajectory's advantage is applied to the tokens the model generates.

Rollout collection. We train with Miles on 4–8 B200 GPUs. Each solver update uses 16 environment groups with 16 trajectories per group. We oversample and discard groups with no reward variation, collecting additional groups until the batch is filled. Unfinished terminal trajectories are carried across updates, preserving their interaction history and live sandbox state. They resume with the updated policy, while stored sampling log-probabilities are retained for previously generated tokens.

Evaluation ranges. Training comparisons match the number of solver updates, not total compute or elapsed time. The full AutoEnvScaling run and Tmax are evaluated through 288 solver updates. The proposer-training ablation is evaluated through 160 updates, so the three-way comparison uses that common endpoint: Terminal-Bench 2.1 scores are 48.9, 44.9, and 40.1 for the full system, Tmax, and the ablation. The ablation peaks at 44.1 at update 96. On Terminal-Bench 4.0, the full system reaches 3.2 at update 192 and retains that score through update 288. Rollout-collection summaries use updates 0–162, the range logged for both methods, with matched hardware and rollout concurrency. Reported retention is total accepted groups divided by total collected groups; collection time is the arithmetic mean of the recorded minutes per update.

G.2 Proposer Validity During Shared-Policy Training

Figure 19 tracks the valid-environment rate of the shared policy acting as proposer during the run in Section 4.3. Both runs start from the Cold-Start checkpoint at 86.8%. With proposer training, the rate dips to 79.6% at step 64, rises above 88% from step 80 onward, and ends at 94.7%. Without proposer training, the same weights still change through solver updates, and the rate falls to 22.4% by step 160, when training collapses.

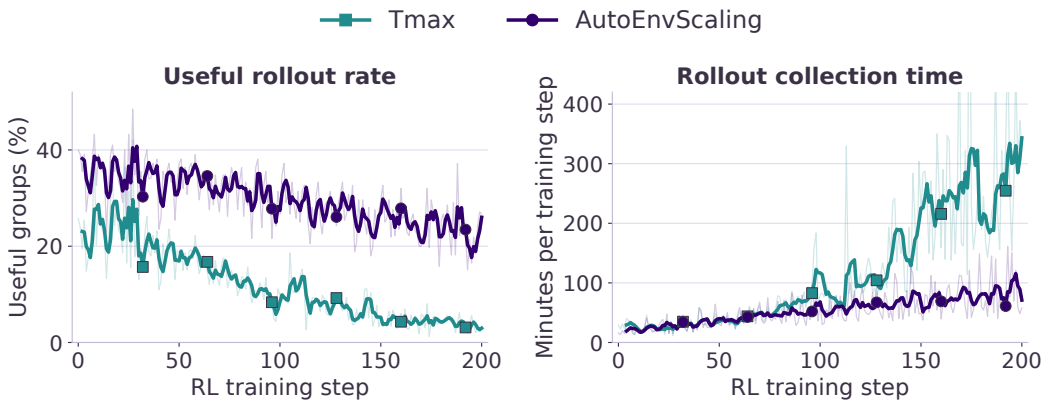


Figure 18: **AUTOENVSCALING retains more rollout groups and collects them faster.** Qwen3.6-35B-A3B trained with AUTOENVSCALING or the static Tmax baseline. **Left:** useful-group rate: groups retained for training as a percentage of all groups collected at each step; both methods retain 16 groups. **Right:** rollout-collection time per step. Faint lines show individual steps; solid lines show 3-step (left) and 5-step (right) trailing means.

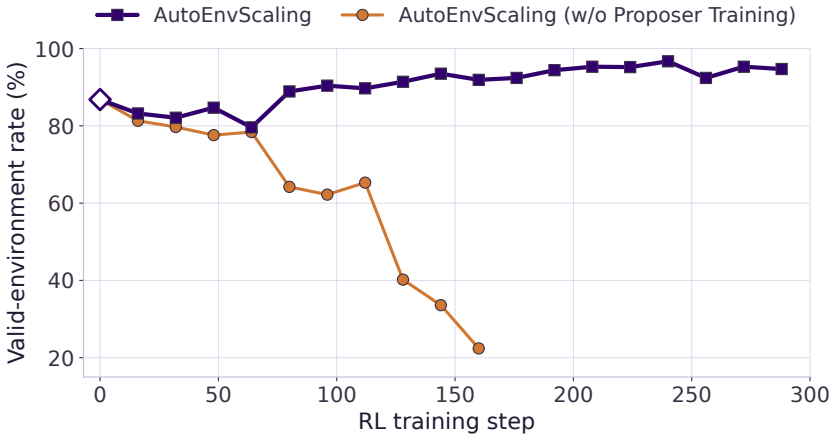


Figure 19: **Proposer training keeps the shared policy's environments valid.** Valid-environment rate of Qwen3.6-35B-A3B acting as proposer during shared-policy training. The open diamond marks the shared Cold-Start checkpoint. The run without proposer training ends at step 160 because training collapsed.

G.3 EVALUATION HARNESS AND RESOURCES

All Terminal-Bench results in this paper use the Pi coding agent in sandboxes with 4 CPUs and 8 GiB of memory. Model cards report higher absolute scores for the same models, for example 40.5 for Qwen3.5-35B-A3B and 51.5 for Qwen3.6-35B-A3B on Terminal-Bench 2.0. Those numbers use the Terminus-2 harness with a 3-hour timeout, in sandboxes with 32 CPUs and 48–64 GB of memory.[2] In our tests, the harness accounts for most of this gap, and the smaller sandboxes for part of the rest. Every comparison in this paper uses the same harness and resources for all methods. Our Tmax reproduction on Qwen3.5-9B (27.2 and 29.2 on Terminal-Bench 2.1) is also close to the 28.8 that Ivison et al. (2026) report with a different harness.

[2] `https://huggingface.co/Qwen/Qwen3.5-35B-A3B`, `https://huggingface.co/Qwen/Qwen3.6-35B-A3B`

G.4 SOLVER-GUIDED CALIBRATION

Admission requires reference reward $r = 1$ and do-nothing reward $r < 0.5$, in addition to valid files and a successful container build. The task review runs `harbor check` with the official Terminal-Bench task-implementation rubric, graded by the proposer's own model. We apply every criterion except `binary_reward`, which requires rewards of exactly 0 or 1, because our tests give partial credit. A task that fails a criterion returns to the proposer for revision. Calibration targets mean reward in $[0.25, 0.75]$ with reward standard deviation at least 0.1. It begins with four solver rollouts and expands to eight when the mean is within 0.15 of a band boundary. Tasks below the band receive one diagnostic rollout with reference-solution hints to guide revision. Each assignment permits up to two revisions. Revised environments repeat the admission checks before entering the training pool.

H COLD-START CONTROLS

Cold-Start fine-tunes Qwen3.6-35B-A3B on 600 high-quality proposer trajectories from Claude Opus 5, DeepSeek-V4-Flash, and Kimi-K3.

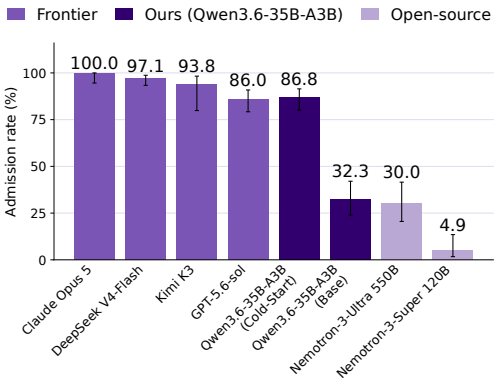


Figure 20: **Cold-Start brings environment generation performance close to frontier models.** Valid-environment rates across frontier models and Qwen3.6-35B-A3B before and after Cold-Start training. Error bars show 95% Wilson intervals; assignment suites vary across models.

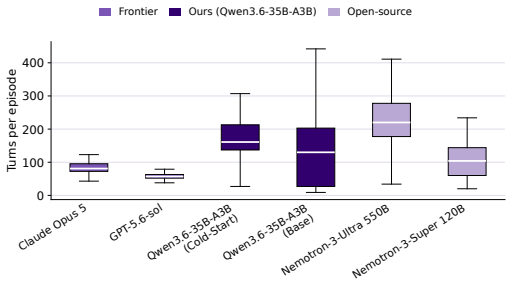


Figure 21: **Proposer episode depth.** Turns per episode for frontier, Cold-Start, and base models. Boxes show the interquartile range and median; whiskers extend to the furthest observations within 1.5 times the interquartile range. Outliers are omitted. Opus uses an earlier assignment suite; the other models use the shared 136-assignment suite.

The comparison in Figure 20 gives the base and Cold-Start Qwen3.6-35B-A3B models and GPT-5.6-sol the same 136 environment-generation assignments. Admission requires a successful image build, a passing reference solution, and a failing do-nothing solution.

We evaluate Qwen3.6-35B-A3B before and after Cold-Start training on environment-generation trajectories. A paired control uses 60 generation assignments, and a separate evaluation measures solving performance on Terminal-Bench 2.1 (Table 13).

Table 13: **Cold-Start improves environment generation without a detected change in solving performance.** Environment acceptance uses the same 60 assignments for both models. Solving uses five attempts on each of 89 Terminal-Bench 2.1 tasks. All values are percentages.

Model	Environment acceptance	TB2.1 avg@5	TB2.1 pass@5
Base	33.3	40.0	59.6
Cold-Start	**76.7**	41.1	66.3

Of the 38 disagreements in task acceptance, 32 favor the Cold-Start model and six favor the base. The mean paired change in solving performance is 1.1 percentage points (95% bootstrap interval: $[-4.9, 7.4]$; paired sign test $p = 0.88$). The pass@5 difference is also not statistically significant ($p = 0.15$).

Training-pool feasibility. In a separate pool-generation experiment, the Cold-Start model supplies 16 tasks with non-degenerate solver rewards (standard deviation ≥ 0.1), enough for one rollout batch. The base model supplies four trainable tasks among 13 evaluated admissions, with one admission unevaluated. Even if that task were trainable, the pool would remain below the 16-task minimum.

I Environment Generation on Two Further Seed Cases

Figure 6 in Section 4.4 shows one seed case. The two below use the same template and the same three proposers.

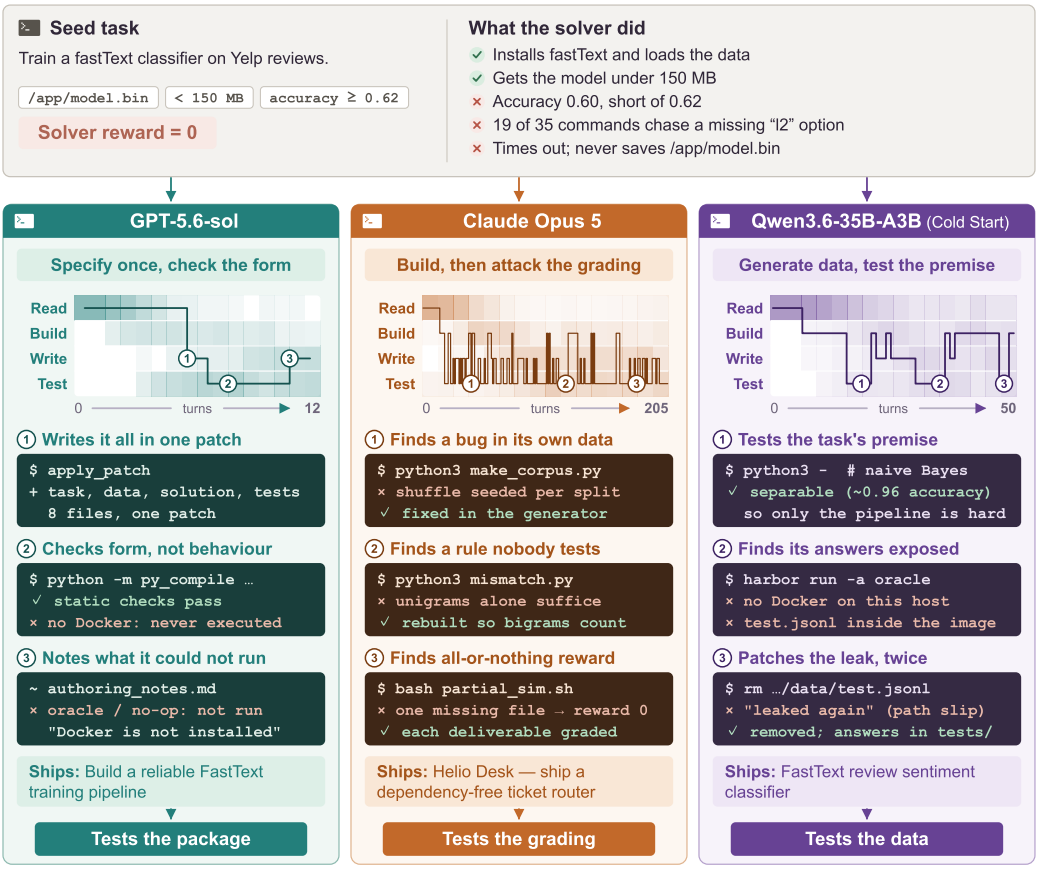


Figure 22: **fastText seed case.** The solver trained a fastText classifier on 650k Yelp reviews, measured 0.60 against a target of 0.62, spent 19 of 35 commands chasing a non-existent option, and timed out without writing the model file. No Docker daemon was available to GPT-5.6-sol, so it ran static checks only. Panels follow Figure 6.

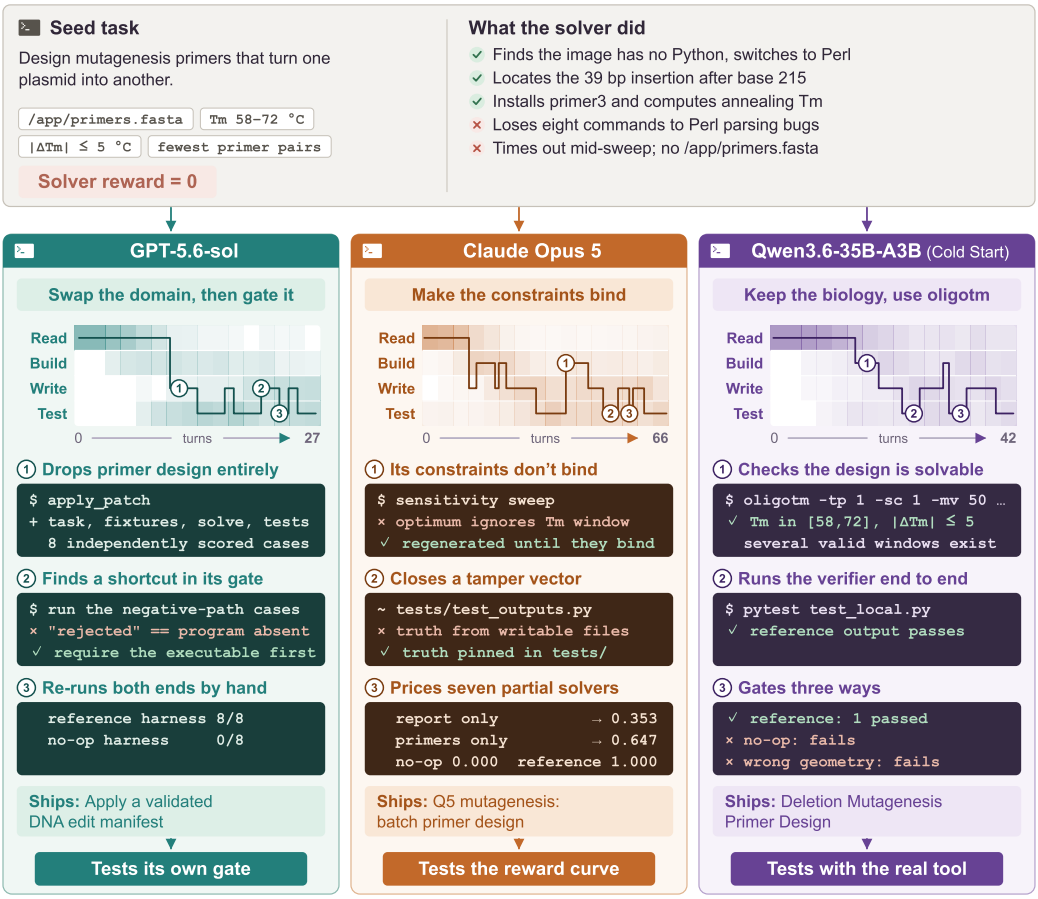


Figure 23: **Primer-design seed case.** The hardest seed task: design Q5 mutagenesis primers under melting-temperature constraints, graded against primer3's `oligotm`. The solver switched to Perl (no Python in the image), located the insertion, computed annealing temperatures, and timed out. GPT-5.6-sol drops primer design for a generic FASTA pipeline; Claude Opus 5 and Cold-Start Qwen3.6-35B-A3B both keep the biology. Panels follow Figure 6.

I.1 Labelling protocol

These three seed cases are the only ones in the sample where all three proposers ship a task without hitting the wall clock, and their seed failures differ: a budget exhausted with nothing produced, a repair loop that timed out mid-fix, and a domain-hard task the solver almost finished.

A turn is a public text step or tool call. Native thinking is excluded. Cold-Start Qwen3.6-35B-A3B's traces carry thinking steps and the other two do not, so its public record is thinner than its actual reasoning. That matters when comparing stated rationale across the three.

All nine drawn paths are labelled by hand, turn by turn, 475 turns in total. The shaded density field behind each path comes from a keyword classifier that agrees with the hand labels on 83 to 90% of turns per episode. We do not use that classifier for the drawn paths. An earlier version of it filed Claude Opus 5's difficulty sweeps under *build*, which inverted the reading.

Episodes use the first sample, with two exceptions: Claude Opus 5 on the fastText case, whose first sample hit the wall clock while waiting on its own reference run, and Cold-Start Qwen3.6-35B-A3B on the primer-design case, whose other samples did not ship. Terminal lines inside the numbered moments are abridged, and a few label an action instead of quoting a command.