EVOTERMINAL: CO-EVOLVING TERMINAL AGENTS THROUGH SCALABLE ENVIRONMENT EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Sustained self-improvement requires not only updating the learner, but keeping its training experiences useful as capabilities change. Executable terminal environments provide a scalable and verifiable substrate for such practice, yet scaling their supply alone cannot keep the training distribution aligned with an evolving learner: informative tasks become redundant, previously unreachable requirements become learnable, and new capability bottlenecks emerge. Crucially, this mismatch is not reducible to difficulty, since a task can remain appropriately challenging while exercising behavior that is no longer limiting. We therefore argue that the executable training distribution should evolve with the learner, adapting both *what to practice* and *how hard*. We introduce EVOTERMINAL, a self-improvement framework that co-evolves terminal agents with their executable training distributions. Its DISCOVER–SHAPE–EVOLVE loop identifies the learner's behavioral frontier from feedback, reallocates practice across capabilities while calibrating difficulty, and realizes these targets as diverse environments that preserve the learning objective. The updated learner reshapes the next distribution, making environment evolution a recurring part of training rather than a one-shot data-generation stage. The same loop supports both model and harness adaptation. On Terminal-Bench 2.0, EvoTerminal raises Qwen3-8B SFT by 12.0 points and Qwen3.6-35B-A3B skill optimization by 5.2 points over their bases, with gains transferring across benchmarks and backbones. Ablations further confirm the importance of feedback refresh and joint capability–difficulty adaptation, underscoring the value of co-evolving training distributions with the learner.

1 INTRODUCTION

Recent terminal agents (Anthropic, 2025; OpenAI, 2025; Google, 2025; OpenCode Team, 2025; SWE-agent Team, 2025; Wang et al., 2025; Merrill et al., 2026) have demonstrated increasingly strong performance on complex command-line tasks, spanning software debugging, system administration, security analysis, and data engineering. In parallel, recent work has substantially expanded executable training environments through procedural generation, environment inversion, and skill-structured synthesis (Gandhi et al., 2026; Lin et al., 2026; Cheng et al., 2026). Together, these advances make terminal-agent training increasingly scalable. Yet scale alone does not guarantee sustained learning value: an environment useful for the current learner may become redundant after an update, while progress can expose a different capability gap. The bottleneck therefore shifts from constructing more executable tasks to keeping the training distribution useful as the learner changes.

A fixed or one-shot environment pool cannot maintain this correspondence. As illustrated in the upper half of Figure 1, some tasks become mastered while others remain beyond the learner's current reach, wasting practice on behaviors that are either already solved or not yet learnable. Prior work has begun to adapt training data through policy-relative selection, difficulty control, and feedback-driven environment synthesis (Zeng et al., 2026; Gu et al., 2026; Shen et al., 2026; Dong et al., 2026). These approaches establish the value of learner-aware adaptation, but useful practice is not determined by difficulty alone. A success rate indicates whether a challenge is appropriately demanding, but not which behavior should be practiced. Two agents with similar success rates may be limited by different stages of planning, repair, or verification, and the dominant frontier of the same learner may shift after each update. The broader question is therefore twofold: *what should the learner practice next, and how hard should that practice be?*

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.

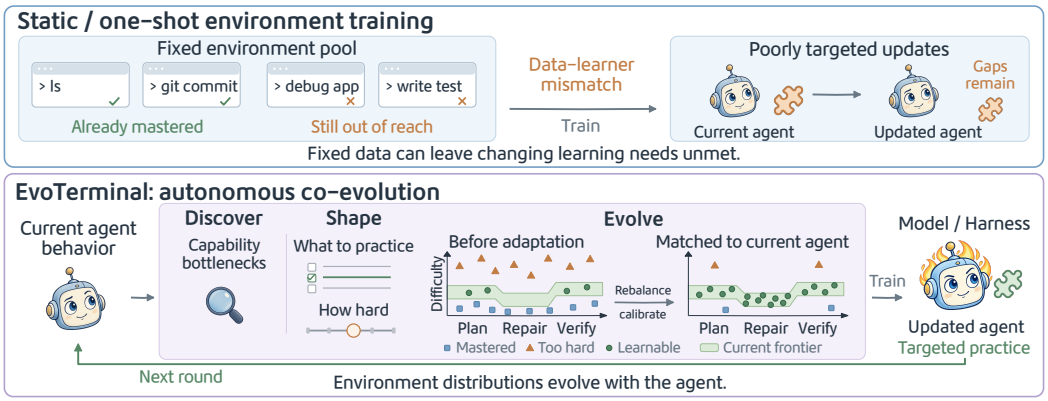


Figure 1: **Executable practice should evolve with the learner.** Top: a fixed pool can allocate practice to mastered or out-of-reach tasks, leaving capability gaps unresolved. Bottom: EvoTerminal discovers the current behavioral frontier, reallocates practice across capabilities, and calibrates difficulty before the next model or harness update.

Our central premise is that the training distribution should *evolve with the learner* along two axes: *what it teaches* and *how much it challenges*. After each learner update, the next batch should reallocate practice toward the current behavioral frontier and calibrate that practice to an appropriate difficulty. Improvement therefore need not make every task harder: as one behavior becomes mastered, practice should shift toward newly exposed requirements. For terminal agents, this adaptation must shape the executable environment itself rather than merely rewrite the instruction. The objective is to keep executable environments aligned with the learner's moving frontier.

We introduce EvoTerminal, a data-centric self-improvement framework that co-evolves terminal agents with executable training distributions. Each round follows Discover–Shape–Evolve: Discover extracts learner-relevant frontier signals from repeated executions and consolidates them into reusable learning targets; Shape reallocates practice across capabilities and calibrates domain-conditioned difficulty; and Evolve realizes these targets through parameterized, lesson-preserving construction of diverse, validated executable environments. Updated learner behavior and progress then reshape the next distribution, closing the environment-evolution loop. Because evolution acts on the training distribution rather than a particular optimizer, the same framework supports both SFT-based model adaptation and frozen-weight skill adaptation of the harness.

Experiments consistently support this co-evolution framework across both update routes. On Terminal-Bench 2.0, EvoTerminal raises frozen-weight Qwen3.6-35B-A3B from 40.0% to 45.2%; SFT gains transfer across benchmarks, and skill gains extend across backbones. Ablations further show the benefits of refreshing learner feedback and jointly adapting capability allocation and difficulty. Our contributions are threefold: **(1) Agent–Environment co-evolution.** We formulate terminal-agent training as a coupled process in which learner behavior reshapes the executable training distribution, which in turn drives subsequent learner updates. **(2) Frontier-driven environment evolution.** We introduce EvoTerminal, which converts repeated-execution feedback into adaptive capability–difficulty curricula and diverse, validated executable environments. **(3) Unified model and harness adaptation.** We show that the same environment-evolution loop supports both SFT-based model adaptation and frozen-weight skill adaptation, with consistent improvements across performance, transfer, and iterative learning.

2 Related Work

Scalable Environment Construction. Recent work has substantially expanded executable training environments through several complementary synthesis paradigms. Procedural and from-scratch pipelines construct verifiable containerized or tool-use environments at scale (Gandhi et al., 2026; Tu et al., 2026; Wang et al., 2026b), while environment inversion and runtime-grounded synthesis derive tasks and trajectories from existing executable states (Lin et al., 2026; Zhu et al., 2026; Wu et al., 2026). Other approaches introduce stronger structural priors through skills, capability require-

ments, or explicit task specifications (Fan et al., 2026; Cheng et al., 2026; Hua et al., 2026; Zhao et al., 2026; Pan et al., 2026). Together, these methods improve the scale, diversity, and controllability of executable practice. EvoTerminal builds on this synthesis substrate but makes the generation target learner-conditioned: the learner's evolving behavioral frontier determines which environment families should receive practice in subsequent rounds.

Adaptive Curricula and Environment Evolution. As the learner changes, the utility of training data becomes non-stationary. Adaptive curriculum methods therefore select or reweight examples according to policy-relative difficulty or expected learning progress (Zeng et al., 2026; Gu et al., 2026). In terminal settings, SETA couples scalable environment synthesis with feedback-driven adaptation of task difficulty and diversity (Shen et al., 2026), while Agent-World combines dynamic task synthesis with continuous training to target evolving capability gaps (Dong et al., 2026). Complementary work on trajectory diagnosis localizes agent failures from execution traces, providing richer behavioral signals than scalar outcomes (Barke et al., 2026). These directions establish the value of learner-aware selection, calibration, and diagnosis. EvoTerminal connects them at the environment-distribution level: behavioral-frontier signals guide *what to practice*, performance feedback calibrates *how hard*, and both jointly determine the executable environments constructed for the next learner update.

Autonomous Agentic Training. A broader line of work closes the improvement loop over different components of the training system. Data-centric approaches iteratively revise data-generation strategies from student or checkpoint feedback (Khan et al., 2025; Meng et al., 2026), while self-play methods generate increasingly informative training problems with limited or no external data (Zhao et al., 2025; Huang et al., 2026; Wei et al., 2026). Other systems evolve the agent or its surrounding training machinery, including agent implementations, training harnesses, and evaluators (Zhang et al., 2026; Chen et al., 2026; Iacob et al., 2026; Wang et al., 2026a). EvoTerminal complements these directions by coupling learner adaptation with evolution of the environment distribution itself: learner behavior shapes the next training environments, which in turn support agent updates.

3 EvoTerminal

3.1 Framework Overview

Problem formulation. We represent a terminal task environment as $x = (u, s_0, c, v)$, where u is the task instruction, s_0 is the initial state containing task files and resources, c specifies the runtime, and v is an executable verifier. At iteration t, the learner $A_t = (M_{\theta_t}, H_{\psi_t})$ consists of a language model M_{θ_t} and an interaction harness H_{ψ_t} with adaptable state ψ_t, such as persistent skills or tool-use procedures. The learner interacts with a batch $\mathcal{D}_t \sim q_t$, where q_t denotes the current distribution over accepted executable environments, and produces rollout evidence $\mathcal{R}_t = \mathrm{Rollout}(A_t, \mathcal{D}_t)$. We use $\mathcal{U}(A, \mathcal{D})$ to denote a generic learner update on training batch $\mathcal{D}$, which may adapt the model, the harness, or both. We additionally maintain a feedback state $\mathcal{F}_t$ summarizing consumed training exposure and learning progress across completed updates.

Environment evolution. The value of an environment distribution is relative to the learner it trains. Let $V(A)$ denote learner performance on a reference development distribution. For a candidate training distribution q under a fixed practice budget, we define its idealized one-step practice utility as

$$\mathcal{G}_t(q) = \mathbb{E}_{\mathcal{D}\sim q}\left[V(\mathcal{U}(A_t, \mathcal{D})) - V(A_t)\right], \tag{1}$$

so that the ideal next distribution satisfies

$$q^*_{t+1} = \arg\max_q \mathcal{G}_t(q). \tag{2}$$

Because $\mathcal{G}_t$ depends on A_t, the optimal training distribution changes as the learner evolves. Directly solving for q^*_{t+1} would require evaluating the counterfactual learning gain of candidate training distributions. EvoTerminal instead approximates this ideal update with a feedback-driven distribution improvement operator,

$$q_{t+1} = \mathcal{T}^{\mathrm{Evo}}_t(q_t, \mathcal{R}_t, \mathcal{F}_t), \qquad \mathcal{T}^{\mathrm{Evo}}_t = \textsc{Evolve} \circ \textsc{Shape} \circ \textsc{Discover}, \tag{3}$$

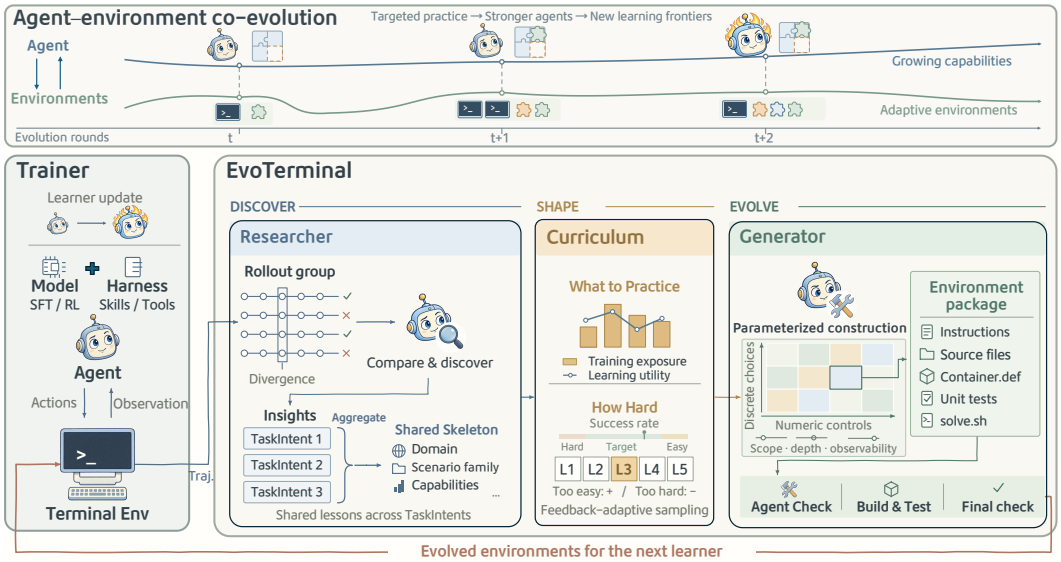


Figure 2: **Overview of EvoTerminal.** Repeated rollouts reveal recurrent behavioral bottlenecks (DISCOVER); the curriculum allocates *what to practice* and *how hard* (SHAPE); and the Generator realizes these targets as validated executable environments (EVOLVE) for the next learner update.

which decomposes the update into identifying learner-relevant practice, shaping its allocation and difficulty, and realizing it as executable environments. Here, $\mathcal{T}_t^{\mathrm{Evo}}$ denotes the feedback-driven environment-evolution operator that maps the current training distribution and learner feedback to the next executable training distribution. As shown in Figure 2, DISCOVER extracts learner-relevant frontier signals from repeated rollouts, distills them into TaskIntents, and consolidates related intents into shared skeletons $S_1, \ldots, S_{K_t}$. Each skeleton defines a reusable environment family, while its TaskIntents specify the behavioral requirements that generated instances must exercise and verify. SHAPE allocates practice across these families and difficulty levels; EVOLVE realizes the resulting construction requests as validated executable environments. The next training distribution is

$$q_{t+1}(x) = \sum_{k=1}^{K_t} \pi_{t+1}(k) \sum_{\ell=1}^{5} w_{t+1}^{d_k}(\ell)\, q_G(x \mid S_k, \ell; q_0)\,, \tag{4}$$

where $\pi_{t+1}(k)$ is the normalized training mass assigned to skeleton S_k, d_k is its domain, and $w_{t+1}^{d_k}(\ell)$ is the domain-conditioned probability of difficulty level $\ell \in \{1, \ldots, 5\}$. For each (S_k, ℓ) request, generation first samples a compatible TaskIntent, preserves its lesson-defining constraints, and varies unconstrained construction choices under the base generator prior q_0. q_G denotes the resulting distribution over accepted realizations. Thus, π_{t+1} controls *what to practice*, w_{t+1}^{d} controls *how hard*, and q_G determines *how that practice is realized* as an environment.

The resulting distribution supplies $\mathcal{D}_{t+1} \sim q_{t+1}$, from which the learner is updated as

$$A_{t+1} = \mathcal{U}(A_t, \mathcal{D}_{t+1})\,. \tag{5}$$

The updated learner provides feedback for the next evolution step, closing the co-evolution loop.

3.2 DISCOVER: TRAJECTORY-CONTRASTIVE FRONTIER ABSTRACTION

Frontier diagnosis. Given rollout evidence $\mathcal{R}_t$, the Researcher extracts learner-relevant frontier signals beyond scalar success or failure. For mixed-outcome tasks, it compares successful and failed executions at their earliest consequential behavioral divergence. For all-failed tasks, it examines the furthest-progressed execution to identify the blocking decision. Consistently solved tasks provide evidence for nearby frontier extensions, while construction and runtime failures are excluded from learner-side analysis. Each actionable signal is distilled into a *TaskIntent* that specifies the target behavior while separating lesson-defining constraints from generation details. It records the target capability, challenge mechanism, observable evidence, required intervention, and verification conditions, turning rollout evidence into a reusable specification for future environment construction.

Cross-task consolidation. The Researcher first groups TaskIntents by diagnosis type, execution stage, and primary capability, then uses semantic similarity to merge redundant intents across tasks. These consolidated groups form shared skeletons $S_1, \ldots, S_{K_t}$. Each skeleton serves as the stable unit on which SHAPE measures learning progress and allocates practice. The skeleton specifies the compatible domain, scenario family, target capability, command surface, and verifier type, while its TaskIntents preserve the lesson-level behavioral targets. The skeleton defines where a lesson can be instantiated, whereas the TaskIntent defines what behavior each instance must elicit and verify. Together, they define the shared behavioral frontier passed to SHAPE.

3.3 SHAPE: CAPABILITY–DIFFICULTY CURRICULUM

What to practice: hierarchical allocation. The Researcher identifies candidate lessons and organizes them into skeletons, while SHAPE determines how a fixed practice budget is allocated without changing their behavioral targets. Because each skeleton is associated with a capability, allocation is hierarchical: SHAPE first distributes training mass across capabilities and then among skeletons within each capability.

Capability allocation balances two signals. *Coverage* favors under-exposed capabilities, with $c_{t,a}^{\mathrm{cov}}$ measuring how little realized training capability a has received. *Learning utility* reflects whether further practice is still productive. For each skeleton S_k, let $\bar{y}_{t,k}^{\mathrm{before}}$ and $\bar{y}_{t,k}^{\mathrm{after}}$ denote mean scores on the same curriculum-development tasks before and after the learner update. We track the smoothed improvement

$$g_{t,k} = \mathrm{EMA}\big(\bar{y}_{t,k}^{\mathrm{after}} - \bar{y}_{t,k}^{\mathrm{before}}\big)\,. \tag{6}$$

Combining $g_{t,k}$ with post-update success and comparison reliability yields a skeleton utility $U_{t,k}$, which favors reliably improving skeletons while downweighting those that are mastered or stalled. For capability a, $\bar{U}_{t,a}$ is the mean utility of its active skeletons with valid progress estimates.

These signals determine the next capability share $\omega_{t+1,a}$. Within each capability, normalized weights $z_{t+1}(k \mid a)$ favor skeletons with larger $U_{t,k}$ while preserving exploration. The resulting training allocation is

$$\pi_{t+1}(k) = \omega_{t+1,a(k)} z_{t+1}(k \mid a(k))\,, \tag{7}$$

where $a(k)$ denotes the primary capability of S_k. Both levels are normalized over active choices. Weights are smoothed across rounds, while exploration priors preserve diversity and retain nonzero mass for skeletons without valid progress estimates. Appendix D gives the scoring functions, update rules, and finite-batch sampling procedure.

How hard: difficulty calibration. For each domain d, the curriculum maintains an L1–L5 distribution $\mathbf{w}_t^d = [w_t^d(1), \ldots, w_t^d(5)]$. Inspired by prior work on optimal learning bands, which favors practice at intermediate success rates (Florensa et al., 2018; Zeng et al., 2026; Guo et al., 2025), we use a shared target-success band $[\tau_L, \tau_H] = [0.20, 0.80]$ across domains. For each observed level, success within the band favors that level, whereas success below or above the band shifts priority toward the adjacent easier or harder level, respectively. These five level-wise scores are confidence-weighted and temporally smoothed into $\mathbf{H}_t^d$, which updates the distribution as

$$\mathbf{w}_{t+1}^d = (1 - \epsilon_d)\,\mathcal{N}\left(\mathbf{w}_t^d \odot \exp(\eta_d \mathbf{H}_t^d)\right) + \epsilon_d \mathbf{1}/5, \tag{8}$$

where $\mathcal{N}$ normalizes the weights over L1–L5, η_d controls the update strength, and ϵ_d preserves exploration across levels. Domain conditioning allows different areas to follow distinct, potentially non-monotonic difficulty frontiers. Together, π_{t+1} and $\mathbf{w}_{t+1}^d$ determine what to practice and how hard each (S_k, ℓ) construction request should be. Appendix D provides the core details.

3.4 EVOLVE: LESSON-PRESERVING WORLD SYNTHESIS

Lesson anchoring. Given a construction request (S_k, ℓ), the Generator samples a compatible TaskIntent and constructs a blueprint. The shared skeleton constrains the environment family, while the TaskIntent anchors the behavioral lesson each realization must preserve. The construction space varies along two axes: *discrete choices* control structural properties such as runtime, toolchain, and artifact pattern, while *numerical controls* regulate properties such as scope, dependency depth, observability, and ambient noise. Lesson-relevant dimensions are conditioned on the skeleton and intent, whereas unconstrained choices follow the base prior q_0. This partial conditioning preserves the learning target while retaining diversity across executable realizations. Details in Appendix C.

Difficulty grounding. The sampled level ℓ grounds abstract difficulty in concrete environment structure. For each scenario family, the Generator biases up to two lesson-relevant construction axes toward the quantile bands associated with ℓ, while sampling the remaining axes from family priors. Lower levels favor local, directly observable problems, whereas higher levels increase scope, coupling, or indirectness. Because these profiles bias rather than fix the parameters, each level still retains substantial variation. For example, a dependency-tracing lesson can progress from a local fault to a cross-component failure by increasing dependency depth and reducing direct observability, while preserving enough evidence for diagnosis.

Generation, Validation and filtering. The Generator materializes the blueprint in stages: it first drafts the task instructions and expected behavior, then writes initial- and final-state tests, and finally produces the source files, container specification, and reference solution. A checklist-guided agent then interacts with the environment, checks lesson consistency and construction quality, and repairs detected defects. The revised candidate must pass three execution checks: the initial environment builds and runs correctly; the verifier rejects the unsolved state without internal errors; and the reference solution makes the verifier accept from a clean initial state. These checks establish executability, non-triviality, and solvability. Finally, an LLM-as-a-judge filters candidates with poor lesson alignment or overall task quality. Only candidates passing all validation gates enter $\mathcal{D}_{t+1}$.

3.5 Learner Update: Model–Harness Adaptation

EvoTerminal can adapt either component of $A_t = (M_{\theta_t}, H_{\psi_t})$ while keeping the other fixed. We instantiate the two routes with supervised fine-tuning (SFT) for model adaptation and SkillOpt-based skill optimization for harness adaptation. Both share the same outer environment-evolution loop, but convert the evolved environments into different learning signals.

Model adaptation via SFT. Before model adaptation, both the current learner and a teacher agent execute the training environments in $\mathcal{D}_{t+1}^{\text{train}}$. Only verifier-passing teacher trajectories from tasks with at least one failed learner rollout are retained as supervised demonstrations; tasks passed in every repeated learner rollout are excluded from SFT. All learner rollouts, including those from excluded tasks, remain available to DISCOVER and SHAPE as behavioral feedback. These filtered demonstrations form the SFT dataset used to update the model while keeping the interaction harness fixed: $A_{t+1} = \left(M_{\theta_{t+1}}, H_{\psi_t}\right)$. Within each outer iteration, SFT is performed on this fixed verified-trajectory set before the environment distribution is refreshed.

Harness adaptation via skills. For harness adaptation, the model remains frozen while the external interaction state is optimized directly from scored terminal rollouts. We use SkillOpt (Yang et al., 2026) to iteratively revise a persistent skill document: $A_{t+1} = \left(M_{\theta_t}, H_{\psi_{t+1}}\right)$. Each inner update evaluates the current skill on a task minibatch, proposes a revision from rollout feedback, and retains it through a fixed development gate. Environment evolution resumes after the inner skill updates.

After either route, the updated learner supplies the behavior and learning-utility signals for the next DISCOVER–SHAPE–EVOLVE round, closing the co-evolution loop. Appendix D.2 details both update procedures and their relation to the outer iteration.

4 Experiments

4.1 Experimental Setup

Base model. We evaluate EvoTerminal through two learner-update routes. In the *SFT route*, we apply full-parameter supervised fine-tuning to Qwen3-8B (Yang et al., 2025) and Qwen3.5-9B (Qwen Team, 2026a). In the *Skill route*, model weights remain frozen while SkillOpt (Yang et al., 2026) updates a persistent skill from scored terminal rollouts. We use Qwen3.6-35B-A3B (Qwen Team, 2026b) for the main skill experiment. Full training details are provided in Appendix B.1.

Benchmarks and evaluation. We use Terminal-Bench 2.0 as the primary benchmark and retain its native evaluation protocol (Merrill et al., 2026). SFT evaluates all 89 tasks; skill learning uses a shared 35/9/45 train–development–test split, with 45 held-out test tasks. Our models use Terminus 2.0 as the harness, with Harbor providing the containerized evaluation infrastructure and executable

task interface (Harbor Framework Team, 2026). We use Gemini-3.6-Flash (Google DeepMind, 2026) for both the Researcher and Generator, while training trajectories for SFT are collected by a DeepSeek-V4-Pro (MAX) teacher agent (DeepSeek-AI et al., 2026). We report our results as mean accuracy ± sample standard deviation over three independent runs. Detailed evaluation configurations are provided in the Appendix B.1.

Baselines. We include general-purpose agents summarized in NexForge (Zhao et al., 2026) and distinguish terminal specialists by their training route. *SFT baselines* comprise CLI-Universe-8B (Hua et al., 2026), trained on 6K Kimi-K2.6 trajectories; Terminal-World-8B (Cheng et al., 2026), trained on 5,723 DeepSeek-V3.2 trajectories; and Meta-Task-14B (Pan et al., 2026), trained with LoRA on 3,221 trajectories from Qwen3.5-397B-A17B. *RL baselines* include SETA-8B (Shen et al., 2026), trained with GRPO on SETA-Env under a 40-epoch configuration, and TMAX-9B (Ivison et al., 2026), trained with DPPO on 15K environments for 500 steps without SFT warm-up. Endless Terminals-8B (Gandhi et al., 2026) uses SFT followed by PPO, starting from an OpenThinker model distilled on 15K GLM-4.6 traces and training on 3,255 Endless Terminals tasks.

4.2 Main Results

EvoTerminal improves SFT performance. Using 2.4K SFT trajectories, EvoTerminal gains 12.0 and 8.2 points over the Qwen3-8B and Qwen3.5-9B bases on Terminal-Bench 2.0 (Table 1). The 8B model approaches Terminal-World-8B (14.6% vs. 15.7%) with fewer than half the trajectories; the 9B model exceeds the reported RL-trained TMAX-9B result (28.5% vs. 27.2%).

Table 1: Terminal-Bench 2.0 pass@1 (%). Left: general agents. Right: terminal specialists and our paired SFT results. General-agent values and scaffold labels are reproduced from NexForge (Zhao et al., 2026). † denotes results reported in the original papers. Our paired results use mean ± SD.

Baselines			
Model	**Params**	**Scaffold**	**TB 2.0 ↑**
Closed-source			
Claude Opus 4.6	–	Claude Code	58.0 ± 2.9†
GPT-5.2	–	Terminus 2	54.0 ± 2.9†
Claude Opus 4.5	–	OpenHands	51.9 ± 2.9†
Gemini 3 Flash	–	Terminus 2	51.7 ± 3.1†
GPT-5.1	–	Terminus 2	47.6 ± 2.8†
Open-weight			
GLM 5	744B	Terminus 2	52.4 ± 2.6†
Kimi K2.5	1T	Terminus 2	43.2 ± 2.9†
MiniMax M2.5	230B	Terminus 2	42.2 ± 2.6†
DeepSeek-V4-Pro	1.6T	official	67.9†

Specialized		
Model	**Scaffold**	**TB 2.0 ↑**
Endless Terminals-8B	Native	6.7†
SETA-8B	CAMEL	10.7†
CLI-Universe-8B	Terminus 2	10.9†
Terminal-World-8B	Terminus 2	15.7†
Meta-Task-14B	Terminus 2	22.5†
TMAX-9B	TMAX	27.2†
Ours		
Qwen3-8B Base	Terminus 2	2.6 ± 0.6
+ EvoTerminal	Terminus 2	**14.6 ± 1.9**
Qwen3.5-9B Base	Terminus 2	20.2 ± 1.1
+ EvoTerminal	Terminus 2	**28.5 ± 1.7**

Cross-benchmark transfer. To verify that EvoTerminal training generalizes beyond Terminal-Bench, we evaluate the same checkpoints on SWE-Bench Verified and AIME 2025 (Table 2). Both backbones improve on both benchmarks, indicating that EvoTerminal induces transferable capability gains rather than benchmark-specific adaptation.

Table 2: **Cross-benchmark transfer.** Parentheses report gains over the corresponding base model.

Model	SWE-Bench Verified	AIME 2025
Qwen3-8B Base	6.0 ± 0.6	68.3 ± 2.5
+ EvoTerminal	**19.0** ± 0.8 (+13.0)	**76.1** ± 2.1 (+7.8)
Qwen3.5-9B Base	42.2 ± 1.1	82.8 ± 2.1
+ EvoTerminal	**48.0** ± 0.8 (+5.8)	**88.1** ± 1.3 (+5.3)

EvoTerminal improves skill training across backbones. Table 3(a) fixes Qwen3.6-35B-A3B and varies only the environments used for skill optimization, with six iterations of 44 train-and-development tasks per condition (264 task uses). EvoTerminal reaches 45.2%, outperforming no

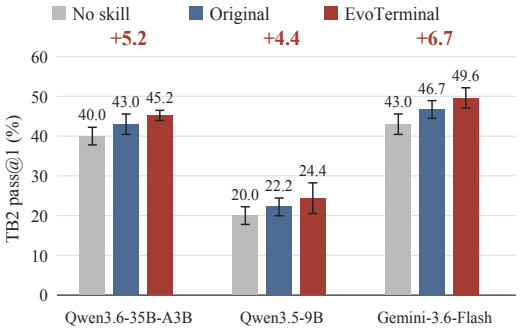
skill by 5.2 points and original environments by 2.2 points. Panel (b) shows Qwen3.5-9B reaching 24.4%, up 4.4 points over no skill and 2.2 points over original environments. Gains also extend to Gemini-3.6-Flash, across model scales and families with frozen weights.

Table 3: Skill training on the 45-task Terminal-Bench 2.0 test split (%). (a) Environment sources on Qwen3.6-35B-A3B; entries are three-run means ± sample standard deviation, and task uses include repeats. (b) Frozen backbones; error bars denote ± one sample standard deviation.

(a) Environment sources

Skill source	Task uses	TB2 ↑
No skill	–	40.0 ± 2.2
Original environments	264	43.0 ± 2.6
Endless Terminals	264	40.7 ± 1.3
TMAX	264	40.0 ± 2.2
SETA	264	42.2 ± 3.8
Ours		
EvoTerminal	264	**45.2 ± 1.3**
Gain over original		**+2.2**

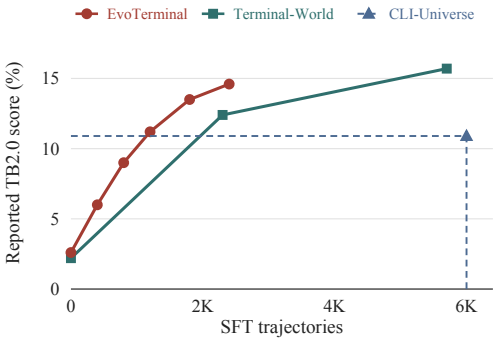
(b) Across backbones



4.3 Analysis

EvoTerminal improves data efficiency and skill optimization. With 2.4K trajectories, Figure 3(a) places EvoTerminal at 14.6%, above Terminal-World's 12.4% at 2.3K. In panel (b), original environments plateau at 43.0, whereas EvoTerminal reaches 45.2 after six updates.

(a) SFT data efficiency (Qwen3-8B)



(b) Skill updates (Qwen3.6-35B-A3B)

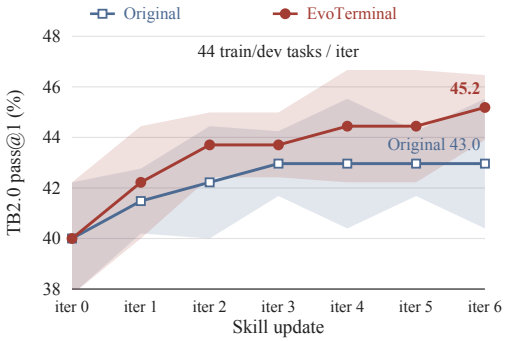


Figure 3: **SFT data efficiency and skill-update dynamics.** (a) Qwen3-8B performance versus SFT trajectories. (b) Qwen3.6-35B-A3B on the fixed test split across six skill updates. Solid lines show three-run means; shading denotes ± one sample standard deviation.

How does the training distribution evolve? Figure 4 shows that EvoTerminal adapts both *what to practice* and *how hard*. Panel (a) displays all eight atomic capabilities and shifts allocation from planning and tool use toward repair, verification, and reflection. Grounding, the primary capability in the software-configuration case (Appendix E), declines from 12.5% to 8.0% globally even as that lineage moves to harder SWE tasks. Panel (b) reveals domain-specific, non-monotonic difficulty adaptation: SWE and debugging move toward harder levels, while security concentrates L2–L3. Appendix B.2 provides the corresponding test-side view of where skill gains appear.

Feedback refresh and online ordering improve SFT. Table 4(a) isolates environment adaptation under the SFT trainer and matched 2.4K-trajectory budget. Static pool samples from fixed distributions, whereas Frozen-feedback adds the initial Research guidance and Shape weights but keeps them fixed. The two offline pools remain close on Terminal-Bench 2.0, suggesting that initial conditioning alone is insufficient. Refreshing feedback improves the resulting pool even after shuffling, while preserving the online order further raises Terminal-Bench 2.0 performance from 25.5 to 28.5. All variants share the Generator, teacher, accepted-trajectory budget, and update schedule.

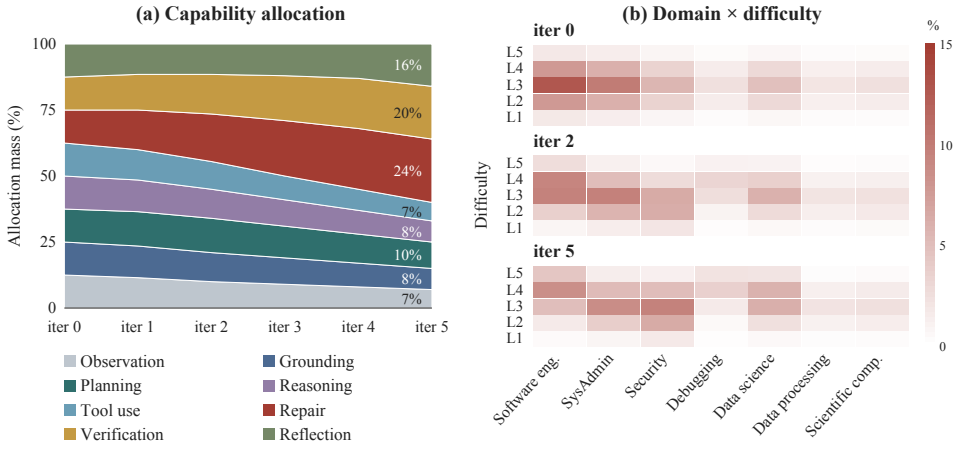


Figure 4: **Evolution of the training distribution.** (a) Controller allocation over eight atomic capabilities across six EvoTerminal rounds, with 44 train–development task uses per round. (b) Joint domain–difficulty allocation at iterations 0, 2, and 5; each snapshot sums to 100%.

Capability and difficulty adaptation are complementary. Table 4(b) isolates the two controls under matched SkillOpt budgets, with Research refresh and generation fixed across conditions. *No curriculum* samples capabilities, skeletons, and L1–L5 uniformly, while the single-control variants adapt only difficulty or capability/skeleton weights. Joint adaptation reaches 45.2, improving over *No curriculum* by 3.7 points and the stronger single-control variant by 2.2 points. This supports adapting both *what to practice* and *how hard*.

Table 4: **Ablations of environment adaptation**. (a) SFT ablations on Qwen3.5-9B evaluated on Terminal-Bench 2.0, separating learner conditioning, feedback refresh, and data ordering under a matched 2.4K-trajectory budget. (b) Skill curriculum ablations on Qwen3.6-35B-A3B isolating capability and difficulty adaptation.

(a) Feedback and order (SFT)

Qwen3.5-9B (SFT)	
Training strategy	**TB2**
Base	20.2
Static pool	24.3
Frozen-feedback	24.0
Evolved pool, shuffled	25.5
EvoTerminal	**28.5**
Gain over frozen	**+4.5**

(b) Curriculum adaptation (Skill)

Qwen3.6-35B-A3B (Skill)			
Curriculum	**Cap.**	**Diff.**	**TB2**
No skill	–	–	40.0
No curriculum	Off	Off	41.5
Difficulty only	Off	On	42.2
Capability only	On	Off	43.0
EvoTerminal	**On**	**On**	**45.2**
Gain over baseline			**+3.7**

5 Conclusion

We presented EvoTerminal, a framework for co-evolving terminal agents with their executable training distributions. Our central insight is that the learning value of practice is learner-relative and non-stationary: as the learner changes, the training distribution should adapt both what to practice and how hard. Discover–Shape–Evolve realizes this principle by turning behavioral frontier signals into curricula and lesson-preserving executable environments. Crucially, environment evolution is not merely difficulty escalation: performance feedback reshapes how hard practice should be, while behavioral feedback reshapes what should be practiced. Experiments across model and harness adaptation support this view, with gains in performance, transfer, and iterative learning, while ablations confirm the value of feedback refresh and joint capability–difficulty adaptation. These results suggest that environment evolution should be a recurring component of the learning loop, rather than a one-shot data-generation stage.

REFERENCES

Anthropic. Claude code: Best practices for agentic coding, April 2025. Anthropic Engineering.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. AgentRx: Diagnosing AI agent failures from execution trajectories, 2026. URL https://arxiv.org/abs/2602.02475.

Guhong Chen, Yingcheng Shi, Yongbin Li, Binhua Li, Xander Xu, Hu Wei, Shiwen Ni, Min Yang, and Jieping Ye. EvoTrainer: Co-evolving LLM policies and training harnesses for autonomous agentic reinforcement learning, 2026. URL https://arxiv.org/abs/2606.03108.

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, 2026. URL https://arxiv.org/abs/2605.20876.

DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. Agent-World: Scaling real-world environment synthesis for evolving general agent intelligence, 2026. URL https://arxiv.org/abs/2604.18292.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, and Lilin Wang. Toward scalable terminal task synthesis via skill graphs, 2026. URL https://arxiv.org/abs/2604.25727.

Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1515–1528, 2018. URL https://proceedings.mlr.press/v80/florensa18a.html.

Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling rl environments for terminal agents, 2026. URL https://arxiv.org/abs/2601.16443.

Google. Gemini cli, 2025. Official Google Gemini CLI repository.

Google DeepMind. Gemini 3.6 Flash: Model card, July 2026. URL https://deepmind.google/models/model-cards/gemini-3-6-flash/.

Zhengyao Gu, Jonathan Light, Raul Astudillo, Ziyu Ye, Langzhou He, Henry Peng Zou, Wei Cheng, Santiago Paternain, Philip S. Yu, and Yisong Yue. Actor-curator: Co-adaptive curriculum learning via policy-improvement bandits for RL post-training, 2026. URL https://arxiv.org/abs/2602.20532.

Jiacheng Guo, Ling Yang, Peter Chen, Qixin Xiao, Yinjie Wang, Xinzhe Juan, Jiahao Qiu, Ke Shen, and Mengdi Wang. Genenv: Difficulty-aligned co-evolution between llm agents and environment simulators, 2025. URL https://arxiv.org/abs/2512.19682.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, jan 2026. URL https://github.com/harbor-framework/harbor.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, Letian Zhu, Xinping Lei, Han Li, Zhiyuan Ma, Zili Wang, Zhaoxiang Zhang, and Jiaheng Liu. CLI-Universe: Towards verifiable task synthesis engine for terminal agents, 2026. URL https://arxiv.org/abs/2606.22883.

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, 2026. URL https://arxiv.org/abs/2508.05004.

Alex Iacob, Andrej Jovanović, William F. Shen, Daniel Burkhardt, Meghdad Kurmanji, Nurbek Tastan, Lorenzo Sani, Niccolò Alberto Elia Venanzi, Ambroise Odonnat, Zeyu Cao, Bill Marino, Xinchi Qiu, and Nicholas D. Lane. The red queen gödel machine: Co-evolving agents and their evaluators, 2026. URL https://arxiv.org/abs/2606.26294.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. TMAX: A simple recipe for terminal agents, 2026. URL https://arxiv.org/abs/2606.23321.

Zaid Khan, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. DataEnvGym: Data generation agents in teacher environments with student feedback, 2025. URL https://arxiv.org/abs/2410.06215.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. CLI-Gym: Scalable CLI task generation via agentic environment inversion, 2026. URL https://arxiv.org/abs/2602.10999.

Fanqing Meng, Lingxiao Du, Qiguang Chen, Ziqi Zhao, Haocheng Lu, Mengkang Hu, and Michael Qizhe Shieh. RSIBench-Data: Benchmarking data-centric research for recursive self-improvement, 2026. URL https://arxiv.org/abs/2607.25886.

Mike A. Merrill et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

OpenAI. Codex cli, 2025. Official OpenAI Codex repository.

OpenCode Team. Opencode: The open source ai coding agent, 2025. Official OpenCode project.

Zhihong Pan, Jiyuan He, Kai Zhang, Yupeng Han, Ze Liu, Yuze Zhao, Yongcong Ye, and Zhaohua Yang. Meta-Task: Turning terminal task synthesis into a terminal task for scalable agent training, 2026. URL https://arxiv.org/abs/2607.27929.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026b. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

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, 2026. URL https://arxiv.org/abs/2607.10891.

SWE-agent Team. mini-SWE-agent, 2025. Official mini-SWE-agent repository.

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.

Beining Wang, Weihang Su, Hongtao Tian, Hao Kong, Tao Yang, Ting Yao, Qingyi Pan, Yueyue Wu, Qingyao Ai, Min Zhang, and Yiqun Liu. Co-evolving LLM evaluators and policies via DynamicRubric, 2026a. URL https://arxiv.org/abs/2607.20083.

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

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.

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, 2026. URL https://arxiv.org/abs/2512.18552.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-scale terminal agentic trajectory generation from dockerized environments, 2026. URL https://arxiv.org/abs/2602.01244.

An Yang et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, Kai Qiu, Yuqing Yang, Dongdong Chen, Xue Yang, and Chong Luo. SkillOpt: Executive strategy for self-evolving agent skills, 2026. URL https://arxiv.org/abs/2605.23904.

Zhiyuan Zeng, Hamish Ivison, Yiping Wang, Lifan Yuan, Shuyue Stella Li, Zhuorui Ye, Siting Li, Jacqueline He, Runlong Zhou, Tong Chen, Chenyang Zhao, Yulia Tsvetkov, Simon Shaolei Du, Natasha Jaques, Hao Peng, Pang Wei Koh, and Hannaneh Hajishirzi. RLVE: Scaling up reinforcement learning for language models with adaptive verifiable environments, 2026. URL https://arxiv.org/abs/2511.07317.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2026. URL https://arxiv.org/abs/2505.22954.

Andrew Zhao, Yiran Wu, Yang Yue, 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, 2025. URL https://arxiv.org/abs/2505.03335.

Jiarong Zhao, Zhikai Lei, Zhiheng Xi, Rui Zheng, Hang Yan, Jie Zhou, Qin Chen, and Liang He. NexForge: Scaling agent capabilities through requirement-driven task synthesis for LLMs, 2026. URL https://arxiv.org/abs/2607.14186.

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, 2026. URL https://arxiv.org/abs/2602.07274.

A Discussion

A.1 AI Use Statement

In this work, we used generative AI tools to assist with research methodology and experimental design, method implementation, analysis and interpretation of results, figure preparation, and manuscript drafting and editing. All AI-assisted work was reviewed by the authors. Executable code and artifacts were tested for correctness where applicable, and cited literature and empirical claims were verified against the referenced sources or project records. We take responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the aid of generative AI.

A.2 Future work.

Our current study leaves several directions for future work. First, although we mainly use supervised fine-tuning to update the learner, the framework is particularly well suited to reinforcement learning: evolved environments provide interactive trajectories together with executable feedback, enabling direct optimization from task outcomes rather than relying only on filtered demonstrations. Second, the evolution process itself may also improve over time. Skills, memories, and reusable procedures accumulated during environment construction could be incorporated into the harness and reused by the Researcher, curriculum controller, and Generator in later iterations. This would extend evolution from tasks and curricula to the mechanisms that drive evolution itself. Finally, we focus on terminal-based tasks in this work. Extending the framework to more general agent environments is an important direction toward studying environment evolution and adaptive curricula for broadly capable agents.

B Experiments

B.1 Experimental Setup and Details

SFT training parameters. We perform full-parameter SFT on verifier-passing teacher trajectories for 2 epochs. The maximum training sequence length is 32,768 tokens for Qwen3-8B and 65,536 tokens for Qwen3.5-9B. Both models use a peak learning rate of 1×10^{-5}, with linear warm-up over the first 10% of optimization steps followed by cosine decay.

Benchmark separation and data provenance. The two update routes use Terminal-Bench 2.0 differently. The Skill route fixes a 35/9/45 train–development–test split of the 89 tasks, approximating a 4:1:5 ratio. The 44 non-test tasks define the *Original environments* condition, while external-source and EvoTerminal conditions match its 35/9 per-iteration budget without using the 45 held-out test tasks. In either learner-update route, EvoTerminal initializes $\mathcal{D}_0$ by sampling from the Generator's default prior rather than from Terminal-Bench 2.0. The SFT route treats all 89 Terminal-Bench 2.0 tasks as a sealed evaluation set, accessed only after training to score fixed Base and SFT checkpoints. Its Discover and Shape updates consume only trajectories and paired development feedback from generated non-benchmark pools. No Terminal-Bench 2.0 task, instruction, trajectory, verifier outcome, or semantic derivative—including a TaskIntent or skeleton—is used to seed, guide, select, or train the SFT environments.

Evolution budgets and feedback. Gemini-3.6-Flash serves as both Researcher and Generator. Each skill condition uses 35 training and 9 development tasks per iteration for six iterations: $6 \times 44 = 264$ task uses. A task use is one task assigned to one iteration; repeated rollouts and inner skill-update steps do not create additional task uses. *Original environments* reuses the same 44 non-test benchmark tasks with fixed train/dev membership in every iteration, without generating new environments. Endless Terminals, TMAX, and SETA each supply 264 distinct source tasks partitioned into six 35/9 train/dev batches; EvoTerminal constructs a fresh 35/9 batch per iteration and evaluates the learner before and after the update on the same nine development tasks. Thus, iteration budgets are matched, while source diversity and reuse follow each condition. The Qwen3-8B data-efficiency study uses up to 2.4K verifier-passing teacher trajectories retained after excluding tasks passed in every pre-update student rollout. Student training-task and development rollouts

supply learner feedback only within the route-specific non-test pools; neither the 45 held-out skill-test tasks nor the 89-task SFT evaluation suite contributes feedback to curriculum construction.

Skill curriculum ablation controls. Table 4(b) varies capability and difficulty adaptation in a 2×2 design. Disabling capability adaptation samples available atomic capabilities uniformly and skeletons within each capability uniformly; disabling difficulty adaptation samples L1–L5 uniformly. All four trained conditions retain Research refreshes, the same Generator and validation gates, and the same initial skill, six-iteration training budget, and evaluation split. Generated environments may differ because the allocation policies differ. No skill is the unoptimized reference rather than a fifth curriculum condition.

Evaluation protocols. We use the same evaluation budgets across model backbones within each benchmark. On Terminal-Bench 2.0, Terminus 2 has no fixed interaction-step cap; both the SFT and Skill routes use 16,384 generated tokens per step and a 128,000-token context window. Each step is one assistant response, which may contain multiple terminal commands. The per-step cap covers the model's reasoning and action output; the context limit covers the task instruction, retained interaction history, tool observations, and current response, rather than cumulative tokens across the entire episode. We average pass@1 over three independent runs, each evaluating every task once: 89 tasks per run for SFT and the same 45 held-out tasks per run for skills. For run-level scores a_1, a_2, a_3, we report $\bar{a} \pm s$, where $\bar{a} = \frac{1}{3}\sum_{r=1}^{3} a_r$ and $s = \sqrt{\frac{1}{2}\sum_{r=1}^{3}(a_r - \bar{a})^2}$ is the sample standard deviation. Table entries, error bars, and shaded bands use the same s to summarize run-to-run variation; ablation tables display means only. Gains are calculated from unrounded means and then rounded to one decimal place. SWE-Bench Verified uses vanilla mini-SWE-agent with Bash-only tools and a 32,768-token context window on all 500 tasks for both backbones. AIME 2025 uses four independent samples per problem on all 30 problems in each of three runs, averaging single-sample correctness within each run. The same run-level mean and standard-deviation calculation applies to both benchmarks. All four checkpoints—Qwen3-8B and Qwen3.5-9B, each before and after EvoTerminal SFT—share a 65,536-token response cap in thinking mode, including reasoning and the final answer. Both Qwen models use top-p 0.95 for AIME; temperature is 0.6 for Qwen3-8B and 1.0 for Qwen3.5-9B. Base and EvoTerminal checkpoints use identical decoding and stopping budgets within each backbone.

B.2 Additional Experimental Analysis

Where do the gains come from? Figure B.1 compares three-run domain scores after skill optimization on original and evolved environments over the 45 held-out Terminal-Bench 2.0 tasks. Both backbones improve in software engineering, security, and debugging. Qwen3.6-35B-A3B gains are concentrated in security and debugging; Qwen3.5-9B also improves in system administration and data science. Other declines modestly for both, while the remaining domains remain stable. Viewed with Figure 4, these results pair gains in software engineering and debugging with harder practice, and security gains with practice concentrated at L2–L3.

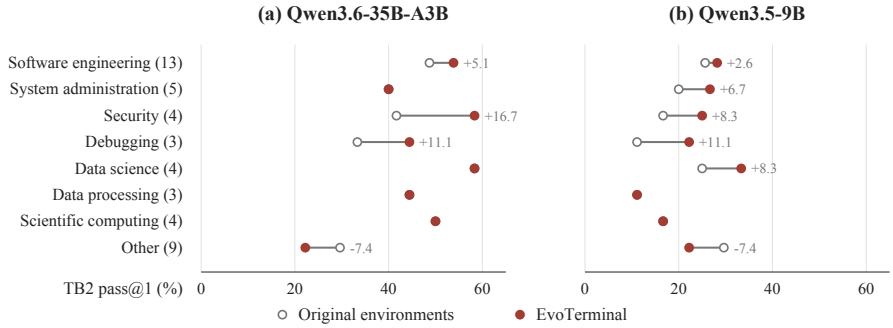


Figure B.1: Three-run mean domain-level skill gains on the 45-task Terminal-Bench 2.0 test split. Open markers denote original-environment skills; filled markers denote EvoTerminal skills.

Environment quality and reliability. We examine whether accepted environments realize their requested learning targets and support reliable execution. The audit samples 200 environments from

the accepted pool across generation rounds, with 40 tasks per requested difficulty level. Table B.1 reports target alignment and teacher rollout outcomes on this balanced sample.

Capability and difficulty alignment. Two human reviewers independently inspect the instructions, initial artifacts, and executable tests without seeing the requested capability, difficulty, or teacher outcomes. They identify the primary capability necessary for a valid solution using the eight atomic categories and assign L1–L5 using the structural profiles in Appendix C.1. The rubric progresses from local work and short dependency chains to cross-component reasoning, distributed diagnosis, and systemic ambiguity. Disagreements are adjudicated before revealing the generation targets. Capability match measures agreement with the requested primary capability; difficulty agreement counts exact matches and ratings within one level. The respective aggregate rates are 89.5%, 77.5%, and 98.0%, indicating that most tasks preserve the target capability and lie at or near the requested structural difficulty.

Teacher solvability and interaction length. DeepSeek-V4-Pro attempts each task from fresh initial states using the same harness and decoding settings. Pass@1 is the fraction of successful rollouts. Mean turns counts actual assistant decisions, including the final response, across both successful and failed attempts. Teacher success falls from 97.5% at L1 to 50.0% at L5, while mean turns rises from 7.8 to 33.5. The fixed teacher therefore encounters increasing practical challenge as the requested level increases.

Table B.1: Environment checks on 200 accepted tasks. Target alignment uses adjudicated blind ratings; Exact and Within 1 compare rated and requested difficulty. Teacher pass@1 uses three independent rollouts per task; mean turns includes successful and failed attempts. Overall aggregates the balanced audit sample, with equal task counts per level.

Requested	Tasks	Target alignment (%)			Teacher rollouts	
level	N	Capability	Exact	Within 1	Pass@1 (%)	Mean turns
L1	40	95.0	85.0	97.5	97.5	7.8
L2	40	92.5	77.5	100.0	92.5	12.6
L3	40	90.0	72.5	97.5	80.8	18.9
L4	40	87.5	72.5	97.5	66.7	26.7
L5	40	82.5	80.0	97.5	50.0	33.5
Overall	**200**	**89.5**	**77.5**	**98.0**	**77.5**	**19.9**

Construction defects. The audit also checks initial-state consistency and agreement between instructions and executable tests. Review flags three tasks with confirmed defects under these checks out of 200 (1.5%): one inconsistent initial fixture and two instruction–test mismatches. These tasks remain in the reported denominators. Construction defects are recorded separately from valid tasks on which the teacher fails to find a solution before the task timeout.

C Generator Parameter Architecture

The Generator turns a construction request (S_k, ℓ) and a compatible TaskIntent into a blueprint $\phi = (\mathbf{c}, \mathbf{z})$. Here $\mathbf{c}$ contains discrete structural choices and $\mathbf{z}$ contains numerical controls. The shared skeleton and TaskIntent constrain the behavioral lesson; the Generator samples the remaining fields within the selected scenario family. This separation preserves the learning target while allowing many executable realizations. Inspired by TMAX's compositional sampling of terminal-task attributes (Ivison et al., 2026), this parameterization makes it easy to encode scenario-family priors and to control selected aspects of generation without fixing the entire instance.

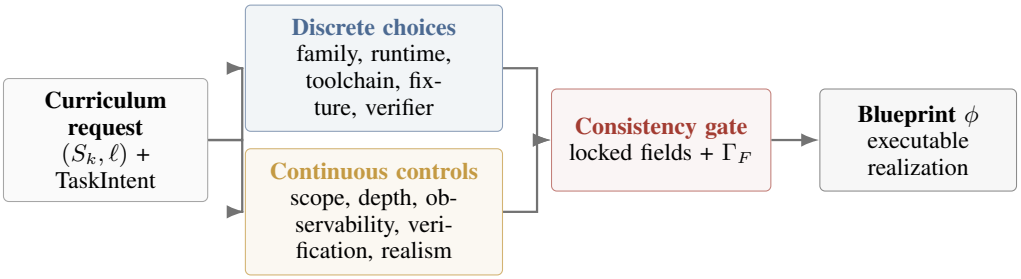


Figure C.1: Generator parameter architecture. The Skeleton and TaskIntent constrain the lesson; discrete and numerical fields are jointly sampled, checked for compatibility, and materialized as an executable blueprint.

C.1 Capability and Difficulty Taxonomy

We distinguish three attributes of a construction request: the *domain* specifies the application setting, the *capability* specifies the behavior to practice, and the *difficulty* specifies the structural challenge. The same capability can appear in several domains and at different levels.

Eight atomic capabilities. The Researcher assigns each TaskIntent one primary capability from Table C.1. For unsuccessful rollouts, this is the earliest behaviorally consequential bottleneck supported by the trajectory contrast; for all-success groups, it is the central capability exercised. Consolidation preserves this attribution as the Skeleton's owner $a(k)$. An environment can require several supporting capabilities without acquiring multiple owners. Family-specific construction tags, such as inspecting source code or preserving backward compatibility, describe finer requirements within this taxonomy; they are not additional atomic categories.

Table C.1: Atomic capability taxonomy. Short labels match the distribution plots.

Capability	Short label	Behavioral boundary
Evidence observation	Observation	Find and read the relevant instructions, artifacts, logs, and state before acting.
Grounding and localization	Grounding	Connect observed symptoms to the file, configuration, component, or artifact that governs them.
Task decomposition and planning	Planning	Organize the objective into feasible, ordered subgoals and a repair–validation plan.
Semantic and logical reasoning	Reasoning	Derive the required behavior from semantics, constraints, invariants, or interface contracts.
Tool execution	Tool use	Use shell, build, database, service, and editing tools correctly to carry out an intended action.
Diagnosis and repair	Repair	Explain the defect and make a targeted correction that preserves the required behavior.
Verification and validation	Verification	Check the resulting behavior and boundary cases, rather than infer success from an intermediate signal.
Adaptive reflection	Reflection	Revise a hypothesis or strategy when new evidence contradicts the current approach.

For example, reading a service log is observation; tracing its symptom to the effective configuration is grounding; correcting the identified defect is repair; checking behavior after a clean restart is verification. The primary label identifies the decisive learning target, rather than every operation in the successful solution.

Five structural difficulty levels. Levels L1–L5 combine causal coupling, localization ambiguity, domain knowledge, design choices, toolchain depth, and the consequences of a change. Table C.2 describes their intended progression. These are structural profiles, not fixed task lengths or learner success rates. The controller's target success band selects useful practice for the current learner; it does not define the levels themselves.

Table C.2: Structural difficulty profiles used by the Generator and target-alignment review.

Level	Profile	Required challenge
L1	Local and self-contained	Visible evidence, a localized intervention, and direct confirmation, while retaining meaningful reasoning.
L2	Small interacting set	Inspect a few related pieces and resolve a short dependency chain using basic domain knowledge.
L3	Cross-component	Trace genuine dependencies across components; the symptom is separated from its cause and a plausible shortcut fails.
L4	Deep and distributed	Resolve competing explanations across interdependent pieces, with indirect evidence, prerequisite troubleshooting, and distributed effects.
L5	Systemic and ambiguous	Experimentally distinguish plausible causes, reconstruct coupled state, and preserve competing invariants across the system.

The selected profile guides the relevant construction axes within a scenario family. Increasing file count or adding routine commands alone does not justify a higher level; the change must make the targeted behavior more demanding.

C.2 Continuous Controls

Each environment samples numerical controls within its scenario family (Table C.3). Scope and depth determine how much state and dependency structure the solver must traverse; observability determines how directly the decisive evidence appears; verifier strength sets the required behavioral coverage; and realism varies plausible background context. Counts use bounded integer ranges, while strengths and densities use continuous intervals.

Table C.3: Numerical control groups. Each row shows representative fields and the aspect of the terminal task they vary.

Axis	Representative controls	Construction effect
Artifact scope	Core artifacts, services, and data size	Varies how many artifacts, services, or records enter the task.
Solution depth	Work units, cross-file locality, and interaction depth	Varies the number and coupling of operations needed for a complete solution.
Observability	Public-signal strength and signal directness	Varies whether decisive evidence is explicit or must be inferred from outputs and state.
Verifier strength	Hidden cases and schema or metric strictness	Varies which behavioral cases and tolerances the final check enforces.
Realism	Ambient-noise density and decoy similarity	Varies plausible background context and distractors without changing the task goal.

Difficulty biases lesson-relevant controls rather than raising every knob together. For scenario family F, let $J(S_k)$ contain the at most two numerical axes most relevant to the lesson. The Generator samples

$$z_j \sim \begin{cases} q_{F,j}^{(\ell)}, & j \in J(S_k), \\ q_{F,j}, & j \notin J(S_k), \end{cases} \qquad |J(S_k)| \leq 2, \tag{C.1}$$

where $q_{F,j}^{(\ell)}$ shifts the family-specific range toward the challenge profile for level ℓ. Other axes retain the family prior, preserving within-level diversity. A difficulty bucket is derived from the realized vector only for analysis and curriculum accounting.

C.3 Discrete Choices

Table C.4 separates three sources of discrete choices. The Skeleton and TaskIntent fix the lesson and verification contract; the scenario family limits compatible implementation forms; and instance sampling varies the task-facing situation and workspace. Choices are resolved in this order, so surface diversity does not overwrite the selected lesson.

Table C.4: Discrete construction choices. The layers separate fixed lesson constraints, family-compatible structures, and sampled instance details.

Layer	**Fields**	**Role**
Skeleton and TaskIntent	Domain, scenario family, primary capability and supporting requirements, command complexity, and verifier contract	Fix the environment family, target behavior, and acceptance conditions.
Family-conditioned	Runtime, language, toolchain, fixture kind, and verifier kind	Select supported structures for realizing that lesson.
Instance-sampled	Persona, work order, artifact pattern, concrete layout, and noise types	Vary task presentation and workspace organization within those constraints.

Explicitly selected fields remain locked. TaskIntent keyword bindings may further constrain open fields, and only the remaining choices are sampled from family-specific priors. Two instances can therefore differ in persona, layout, fixture, or toolchain where the family permits, while preserving the same target capability and verification conditions.

C.4 Cross-Parameter Consistency Control

The parameters are not sampled independently. Each scenario family F defines a compatibility set Γ_F over the joint blueprint. A candidate proceeds only when

$$(\mathbf{c}, \mathbf{z}) \in \Gamma_F, \tag{C.2}$$

which checks domain–family, family–runtime, family–fixture, family–verifier, fixture–verifier, and quantitative realism constraints. For example, a service-backed verifier requires a compatible runtime and fixture, while verifier breadth must remain plausible for the sampled artifact scope.

Resolution follows a fixed priority: the Skeleton and TaskIntent constraints come first, followed by explicitly locked fields, the scenario-family registry, and stochastic sampling. A conflicting sampled field is resampled from its compatible set; a conflict among locked fields rejects the blueprint. Once the joint parameters are consistent, the Generator materializes the environment. The subsequent initial-state, unsolved-state, and solved-state checks then verify that the realized task, container, tests, and reference solution satisfy the blueprint's contract.

D METHOD DETAILS

This appendix records the implementation defaults, end-to-end iteration, and the controller details underlying Section 3.3.

D.1 DEFAULT IMPLEMENTATION PARAMETERS

Table D.1 lists the principal controller settings used in our experiments.

Table D.1: Core settings for capability allocation and difficulty control.

Component	**Parameter**	**Default**
Allocation	Coverage–utility mixture λ	0.30
Difficulty	Target success range $[\tau_L, \tau_H]$	$[0.20, 0.80]$
	Update strength η_d; level exploration ϵ_d	1.50; 0.10

D.2 ITERATION PSEUDOCODE

Outer iterations and inner update steps. An *iteration* (iter) comprises one environment-evolution round and the learner update on its accepted pool. Within iter t, the pool is held fixed while K_t^{upd} inner *update steps* change either model weights or the skill. An SFT step is one optimizer update on an effective minibatch; a SkillOpt step is one rollout–analysis–revision–selection cycle on a task minibatch. Terminal interaction steps occur inside each rollout and use a separate budget. Environment evolution resumes after all K_t^{upd} update steps finish.

Environment-pool lifecycle. The seed pool $\mathcal{D}_0$ exposes the initial learner behavior. At iteration t, $\mathcal{D}_t$ is used to collect $\mathcal{R}_t$, after which EVOLVE constructs a fresh accepted pool $\mathcal{D}_{t+1}$ and splits it once into train and development partitions. The train partition supplies filtered teacher trajectories for SFT or rollout minibatches for SkillOpt. Each iteration refreshes the development partition and recomputes A_t's accuracy on it; A_{t+1} is then evaluated on the same tasks for paired progress. $\mathcal{D}_{t+1}$ replaces $\mathcal{D}_t$ as the rollout pool for the next iteration rather than being appended to a cumulative replay buffer. The feedback state $\mathcal{F}_t$ retains exposure, outcome, and paired-progress summaries across iterations, while the held-out test set remains outside this loop.

For SFT, let n_t^{SFT} be the number of accepted teacher trajectories, E_t^{SFT} the number of epochs, and B_t^{SFT} the effective global minibatch size. The number of model updates is $K_t^{\mathrm{SFT}} = E_t^{\mathrm{SFT}} \lceil n_t^{\mathrm{SFT}} / B_t^{\mathrm{SFT}} \rceil$ when the final partial minibatch is retained. For skill learning, let n_t^{tr} and n_t^{dev} denote the train and development task counts, and let B_t^{skill} be the train-task minibatch size. Then $K_t^{\mathrm{skill}} = \lceil n_t^{\mathrm{tr}} / B_t^{\mathrm{skill}} \rceil$ revision-and-selection steps are performed. Local step $k \in \{1, \ldots, K_t^{\mathrm{upd}}\}$ has global index $g = \sum_{r<t} K_r^{\mathrm{upd}} + k$; the learner state carries forward between iterations.

Algorithm D.1 EVOTERMINAL: outer environment-evolution loop

Input: learner A_0, accepted seed pool $\mathcal{D}_0$, route $r \in \{\mathrm{SFT}, \mathrm{Skill}\}$, iterations T, accepted-task budget N.

1 Initialize feedback state $\mathcal{F}_0 \leftarrow \emptyset$.
2 **for** iter $t = 0, \ldots, T-1$ **do**
3 Execute A_t on $\mathcal{D}_t$ to collect learner rollouts $\mathcal{R}_t$.
4 DISCOVER: analyze $\mathcal{R}_t$ and obtain TaskIntents and skeletons.
5 SHAPE: use $\mathcal{R}_t, \mathcal{F}_t$ to allocate capability and difficulty.
6 EVOLVE: generate and validate N fresh accepted environments; split the pool once into $\mathcal{D}_{t+1}^{\mathrm{train}}$ and $\mathcal{D}_{t+1}^{\mathrm{dev}}$.
7 Evaluate A_t on $\mathcal{D}_{t+1}^{\mathrm{dev}}$ to obtain the before-update score.
8 **if** $r = \mathrm{SFT}$: $A_{t+1} \leftarrow \mathrm{SFTUPDATE}(A_t, \mathcal{D}_{t+1}^{\mathrm{train}})$ (Algorithm D.2).
9 **else**: $A_{t+1} \leftarrow \mathrm{SKILLUPDATE}(A_t, \mathcal{D}_{t+1})$ (Algorithm D.3).
10 Reevaluate A_{t+1} on the same development tasks; obtain $\mathcal{F}_{t+1}$ by appending exposure, learner outcomes, and paired progress to $\mathcal{F}_t$.
11 **return** A_T.

Algorithm D.2 SFTUPDATE: inner model-parameter updates

Input: $A_t = (M_{\theta_t}, H_{\psi_t})$ and the round's training pool.

1 Run the current learner and fixed teacher on $\mathcal{D}_{t+1}^{\text{train}}$; score their rollouts with task verifiers.
2 Retain all learner rollouts for feedback; form $\mathcal{Z}_t$ from verifier-passing teacher trajectories only for tasks with at least one failed learner rollout.
3 Set $\theta \leftarrow \theta_t$, keep ψ_t fixed, and initialize local update counter $k \leftarrow 0$.
4 **for** epoch $e = 1, \ldots, E_t^{\text{SFT}}$ **do**
5 Shuffle $\mathcal{Z}_t$ and partition it into effective minibatches of size B_t^{SFT}.
6 **for each** minibatch $\mathcal{B}$ **do**
7 Compute next-token loss on teacher assistant responses, conditioning on instructions and interaction history.
8 Accumulate gradients and take one optimizer update of θ; advance the learning-rate schedule and $k \leftarrow k + 1$.
9 **return** $A_{t+1} = (M_\theta, H_{\psi_t})$ after K_t^{SFT} updates.

Algorithm D.3 SKILLUPDATE: inner skill-document updates

Input: $A_t = (M_{\theta_t}, H_{\psi_t})$, n_t^{tr} train tasks, and n_t^{dev} development tasks.

1 Freeze θ_t, initialize skill $\psi \leftarrow \psi_t$, and score it on $\mathcal{D}_{t+1}^{\text{dev}}$.
2 Shuffle the train tasks into K_t^{skill} minibatches $\mathcal{B}_1, \ldots, \mathcal{B}_{K_t^{\text{skill}}}$ of size B_t^{skill}.
3 **for** local update step $k = 1, \ldots, K_t^{\text{skill}}$ **do**
4 Run the learner with current skill ψ on $\mathcal{B}_k$; score the rollouts with task verifiers.
5 Analyze rollout successes and failures; aggregate actionable changes to the skill.
6 Apply the proposed revision to produce candidate skill $\widetilde{\psi}$.
7 Evaluate $\widetilde{\psi}$ on the same development set; retain the better-scoring skill, keeping ψ on a tie.
8 Carry the selected skill into the next update step.
9 **return** $A_{t+1} = (M_{\theta_t}, H_\psi)$ after K_t^{skill} revision-and-selection steps.

Feedback timing and notation. The accepted pool is fixed throughout an iteration. Filtered teacher trajectories provide SFT supervision; pre-update learner rollouts, including those from all-pass tasks excluded from SFT, and matched development scores provide environment-evolution feedback. Development tasks select skills and measure progress, while the held-out test set is reserved for reporting. In this synchronous presentation, one completed iteration supplies one feedback interval j to the controller; the following equations also allow finer intervals. Histories remain specific to the learner and training route. Below, $\mathcal{N}$ normalizes a nonnegative vector to sum to one, returning a uniform vector when its sum is zero; $[z]_+ = \max(z, 0)$, and $\mathrm{softmax}$ uses the indicated index set.

D.3 CAPABILITY COVERAGE AND LEARNING UTILITY

Coverage from consumed practice. Each skeleton S_k has one owner capability $a(k)$ from a fixed set of $C = 8$ capabilities defined in Appendix C.1: evidence observation, grounding/localization, task decomposition/planning, semantic/logical reasoning, tool execution, diagnosis/repair, verification/validation, and adaptive reflection. Let $\widetilde{E}_{t,a}$ be the normalized exposure summary for capability a at round t, computed from the training resource actually consumed rather than from proposed environments. It combines total and recent exposure. Coverage is the normalized deficit below equal exposure:

$$c_{t,a}^{\text{cov}} = \mathcal{N}\Big([\mathbf{1}/C - \widetilde{\mathbf{E}}_t]_+\Big)_a . \tag{D.1}$$

The accounting unit follows the trainer's records: consumed training tokens when available and consumed tasks otherwise.

Utility from completed learner updates. Utility is computed only after a real learner update and from matched development comparisons. For skeleton S_k, let $y^-_{t,k}$ and $y^+_{t,k}$ be its before/after scores, $p_{t,k} = y^+_{t,k}$ its post-update success, and $g_{t,k}$ the exponentially smoothed gain. We summarize the useful learning signal as

$$\begin{aligned} g_{t,k} &= \mathrm{EMA}(y^+_{t,k} - y^-_{t,k}), \\ U_{t,k} &= \underbrace{r_{t,k}}_{\text{evidence weight}} \Bigg[\underbrace{g_{t,k}}_{\text{observed gain}} + 0.20 \, \underbrace{4p_{t,k}(1-p_{t,k})}_{\text{mid-success priority}} \Bigg] - \underbrace{P_{t,k}}_{\text{state penalty}} . \end{aligned} \tag{D.2}$$

The first valid interval initializes $g_{t,k}$ with its observed gain; subsequent valid intervals use 0.70 times the previous EMA plus 0.30 times the new gain. For the latest valid feedback interval, let $n^{\mathrm{eff}}_{t,k} = \sum_j \min(n^-_{t,k,j}, n^+_{t,k,j})$, where j ranges over valid before/after comparisons on the same development tasks, and $n^-_{t,k,j}$ and $n^+_{t,k,j}$ are their sample counts. The minimum limits each comparison to the sample support available on both sides. We set $r_{t,k} = n^{\mathrm{eff}}_{t,k}/(n^{\mathrm{eff}}_{t,k} + 8)$. The constant 8 sets the shrinkage scale: eight paired samples give $r_{t,k} = 0.5$.

The intermediate-success term favors learnable rather than saturated tasks. Using the latest post-update success $p_{t,k}$ and smoothed gain $g_{t,k}$, the state penalty is

$$P_{t,k} = \begin{cases} 0.25, & p_{t,k} \geq 0.85 \text{ and } |g_{t,k}| \leq 0.01 \quad \text{(mastered)}, \\ 0.25, & p_{t,k} \leq 0.20 \text{ and } g_{t,k} \leq 0.01 \quad \text{(stalled)}, \\ 0, & \text{otherwise.} \end{cases} \tag{D.3}$$

These thresholds and the 0.25 penalty are fixed controller defaults, not quantities estimated from the development set. The penalty exceeds the maximum 0.20 mid-success bonus, so a mastered or stalled skeleton with negligible gain cannot be prioritized solely by that bonus. Here *stalled* specifically means low success without gain above the 0.01 tolerance; flat gain at intermediate success does not trigger this penalty. A skeleton without any valid comparison has $U_{t,k} = 0$.

D.4 Hierarchical Allocation and Finite-Batch Sampling

From capability priorities to skeleton weights. For each capability a, let $\mathcal{S}_{t,a}$ be its skeletons active for allocation and $\mathcal{V}_{t,a} \subseteq \mathcal{S}_{t,a}$ those with valid progress comparisons. Their mean utility is $\bar{U}_{t,a} = |\mathcal{V}_{t,a}|^{-1} \sum_{k \in \mathcal{V}_{t,a}} U_{t,k}$ when $\mathcal{V}_{t,a} \neq \emptyset$, and 0 otherwise. We then set $c^{\mathrm{gain}}_{t,a} = \mathrm{softmax}_a(\bar{U}_{t,a})$. The core capability target combines exposure coverage and observed learning utility:

$$\widehat{\omega}_{t+1,a} = (1 - \lambda_t) c^{\mathrm{cov}}_{t,a} + \lambda_t c^{\mathrm{gain}}_{t,a}, \tag{D.4}$$

where $\lambda_t = \lambda$ when an active skeleton has a valid progress comparison, and 0 otherwise. The implementation smooths $\widehat{\omega}_{t+1}$ across rounds and adds a small uniform floor to obtain ω_{t+1}. Within each capability, utility gives the core distribution over $\mathcal{S}_{t,a}$:

$$\widehat{z}_t(k \mid a) = \frac{\exp(U_{t,k})}{\sum_{k' \in \mathcal{S}_{t,a}} \exp(U_{t,k'})}, \qquad \pi_{t+1}(k) = \omega_{t+1,a(k)} z_t(k \mid a(k)). \tag{D.5}$$

Here z_t is $\widehat{z}_t$ after a small uniform-floor stabilization that preserves its normalization within each capability.

Realizing a finite batch. Only skeletons with an associated TaskIntent enter the sampling pool. The targets $\pi_{t+1}(k)$ are renormalized over available capabilities. To realize a batch of N requests, the sampler draws sequentially from these targets and downweights a capability, capability–domain pair, or skeleton once its realized count exceeds N times its target mass. This soft correction approximates the continuous allocation without imposing brittle hard quotas; resumed sampling retains the counts already drawn.

D.5 Domain-Conditioned Difficulty

Success scores and neighboring levels. Within each domain and feedback interval, repeated rollouts are averaged within each task; task groups then contribute equally to each level's success rate s_ℓ. The count n_ℓ therefore counts task groups, not individual rollouts. SFT demonstrations are excluded from learner difficulty feedback. With target success bounds τ_L, τ_H, define

$$F(s) = \begin{cases} -(\tau_L - s)/\tau_L, & s < \tau_L, \\ 1 - \dfrac{|s - (\tau_L + \tau_H)/2|}{(\tau_H - \tau_L)/2}, & \tau_L \le s \le \tau_H, \\ -(s - \tau_H)/(1 - \tau_H), & s > \tau_H, \end{cases} \tag{D.6}$$

The score is positive for useful intermediate success and negative when the level is too hard or too easy. For an observed level ℓ in domain d and feedback interval t, let $f^d_{t,\ell} = c^d_{t,\ell} F(s^d_{t,\ell})$, where $c^d_{t,\ell} = \min\{1, n^d_{t,\ell}/3\}$; set $f^d_{t,\ell} = 0$ when the level is unobserved. This shrinks scores supported by only a few task groups. If a level is too easy or too hard, half the magnitude of its negative score also favors the adjacent harder or easier level, respectively. Denote the resulting scores by $\tilde{f}^d_{t,\ell}$. This shifts practice toward the learner's boundary while retaining the evidence that the source level is currently unproductive.

Smoothed probability updates. We combine each adjusted score with its running average:

$$m^d_{t,\ell} = \beta m^d_{t-1,\ell} + (1 - \beta)\tilde{f}^d_{t,\ell}, \qquad H^d_{t,\ell} = \alpha \tilde{f}^d_{t,\ell} + (1 - \alpha) m^d_{t,\ell}. \tag{D.7}$$

The average starts at zero; $\beta = 0.6$ retains past evidence and $\alpha = 0.7$ emphasizes the current frontier. All hyperparameters in the curriculum controller were chosen heuristically without a dedicated tuning search; their exact values are not central to the framework. Thus, $\mathbf{H}^d_t$ combines where practice is productive with where it should move next, without reacting too strongly to a sparse interval. The level distribution is then updated by

$$\mathbf{w}^d_{t+1} = (1 - \epsilon_d)\mathcal{N}\big(\mathbf{w}^d_t \odot \exp(\eta_d \mathbf{H}^d_t)\big) + \epsilon_d \mathbf{1}/5. \tag{D.8}$$

Here η_d sets the update strength and ϵ_d keeps every level reachable. Domains start from the same L3-centered prior. For each round, one level is drawn per skeleton from its domain distribution and then fixed for all instances of that skeleton.

E Case Study

E.1 Discover: Isolating a Reusable Failure Mechanism

We trace a software-engineering lineage centered on effective runtime configuration. The starting task presents a Python service whose `/health` endpoint succeeds while `/v1/report` returns a stale schema after restart. Three solver paths share the same initial state and verifier. Table E.1 contrasts their first decisive actions.

Table E.1: Trajectory contrast used to derive the case-level TaskIntent.

Rollout	Evidence used	Intervention	Observed boundary
Path 1, fail	Handler code and visible response log	Patches response serialization and restarts the service	Report remains stale; final verification fails.
Path 2, fail	Generated manifest and warm-process state	Edits the generated manifest directly and performs a warm reload	Output changes once, then reverts after a clean restart.
Path 3, pass	Process arguments, service environment, generator, and manifest provenance	Repairs the authoritative source, regenerates the manifest, and cold-restarts the service	Health and report contracts both hold after reconstruction.

The decisive distinction is whether the solver localizes the authoritative state before repairing downstream effects. This makes grounding/localization the primary TaskIntent capability, with diagnosis/repair and verification as supporting steps (Appendix C.1). The learning target is to identify which state governs the symptom before deciding how to repair it.

Discover **TaskIntent: effective-configuration localization before repair**

Construction objective. Build tasks in which a visible runtime symptom is governed by an upstream authoritative state and at least one derived layer.
Causal chain. Public symptom → runtime provenance → authoritative source → derived state → clean-state behavior.
Evidence path. Expose process arguments, service metadata, generation commands, and reproducible endpoint behavior without naming the defective layer.
Required intervention. Modify the authoritative source, regenerate dependent artifacts, and restart affected services to restore consistent behavior.
Discriminating verification. Reject downstream-only edits, direct edits to generated artifacts, stale outputs, and warm-state success; accept equivalent repairs that survive reconstruction and a clean restart.
Generation contract. Preserve causal localization and clean-state verification; vary the domain surface, runtime, artifacts, services, and symptom.

Source evidence	Reusable abstraction	Generation freedom
Handler edits fail; a generated file works only until restart; process metadata identifies a higher-precedence source.	Localize the governing state, intervene upstream, reconstruct derived state, and verify persistence.	Change services, tools, files, symptoms, and narratives while preserving the provenance chain and clean-state check.

E.2 SHAPE: MOVING THE PRACTICE FRONTIER

Within one generated environment, the selected TaskIntent remains fixed. Across iterations, new rollout evidence motivates a related TaskIntent as the learner's behavioral frontier moves. In this lineage, the invariant is to localize authoritative state, intervene upstream, and verify from a clean state; successive intents add precedence resolution and then multi-consumer consistency.

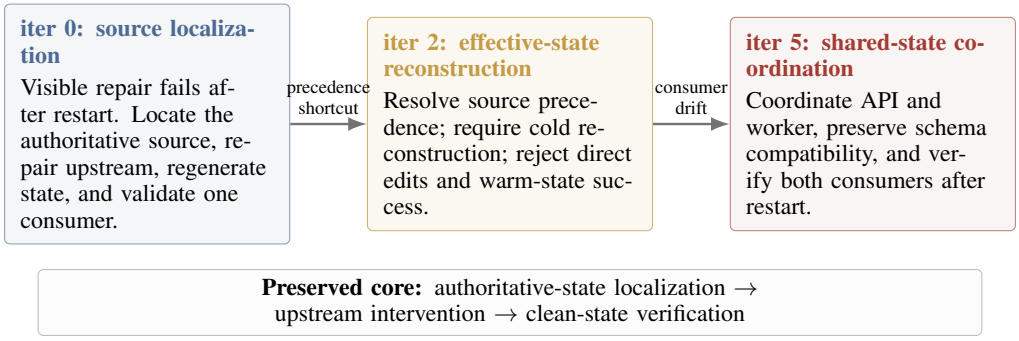


Figure E.1: TaskIntent progression across one capability lineage. Each transition addresses a newly exposed shortcut while preserving the behavioral core.

After DISCOVER selects a TaskIntent and shared skeleton, SHAPE allocates practice to its primary capability and skeleton, then samples a level from the domain-conditioned difficulty distribution. The Generator instantiates the corresponding environment. Table E.2 shows realizations at three sampled levels. The controller snapshots below use the distribution values underlying Figure 4: the first three rows are global atomic-capability shares, whereas SW L4–5 is the share of L4–L5 *within software engineering*, not a global share or the level assigned to this one lineage.

Table E.2: Environment realizations under successive TaskIntents and sampled difficulty levels.

Round	Level	Environment realization	Shallow behavior rejected
iter 0	L3	Source–generator–manifest dependency governs one service	Editing the visible handler
iter 2	L4	Competing configuration sources and a service-manager override obscure effective state	Editing the derived file or using a warm reload
iter 5	L5	Ambiguous provenance and partial reload couple two consumers with compatibility constraints	Repairing one process, hard-coding one response, or skipping reconstruction

SHAPE: controller snapshots

	iter 0	iter 2	iter 5
Grounding	12.5%	11.0%	8.0%
Repair	12.5%	18.0%	24.0%
Verification	12.5%	15.0%	20.0%
Software engineering L4–5	29.9%	46.0%	65.0%

The progression changes the required reasoning rather than the amount of routine work. At iter 0, one provenance trace identifies the source. At iter 2, the solver must distinguish source configuration, a generated manifest, and a service-manager override. At iter 5, it must separate precedence errors from stale consumer state, coordinate two consumers, and verify their shared contract after reconstruction. The global grounding share decreases from 12.5% to 8.0% in Figure 4(a), while software-engineering practice shifts toward L4–L5 in panel (b): capability priority and within-domain structural difficulty are separate curriculum decisions.

E.3 EVOLVE: REPRESENTATIVE ITER 5 ENVIRONMENT

EVOLVE **Software configuration recovery, L5**

Public task core. Restore consistent versioned report schemas across an API and an asynchronous worker. Both components must remain healthy after a cold restart, and existing client fields must remain backward compatible.
Initial state and signal. Health checks pass, but the API and worker emit different schema versions after the supplied reproduction command. Service status, process arguments, reload logs, and the generation entry point are public.
Hidden construction mechanism. Two candidate configurations exist; a service-manager override selects the source from which the shared manifest is generated. Both components consume that manifest, but a warm reload refreshes only the API.
Completion boundary. The final verifier cold-starts both components, checks multiple report shapes and existing client fields against the public schema contract, and rejects stale, partial, single-process, and fixture-specific repairs.
Package validation. Checklist review precedes the three execution checks: the initial-state check accepts the usable but unsolved environment, the final verifier rejects the baseline, and the reference repair passes after reconstruction. The independent Judge confirms lesson fidelity.
Targeted solver behavior. Trace configuration provenance, repair the selected source, regenerate the manifest, restart both consumers, and validate the report and compatibility cases.

This lineage connects the complete loop: rollout contrast identifies the capability boundary, SHAPE allocates practice and samples domain-conditioned difficulty, and EVOLVE constructs a world that preserves the lesson while rejecting the earlier shortcut.

Preserved core	Evolving construction	Required solver transition
Localize authoritative state, intervene upstream, and verify a clean reconstruction.	Add provenance layers, weaker direct observability, coupled consumers, restart semantics, and stronger negative cases.	Symptom editing → provenance tracing → coordinated repair and persistent validation.

E.4 Skill Evolution: From Goal Reminder to Conditional Procedure

We separately trace a general terminal skill ψ under the harness-adaptation route. With the model frozen, SkillOpt proposes bounded revisions from scored rollouts and retains a revision only when it improves the development gate (Yang et al., 2026). Figure E.2 follows the same skill at iter 0, 3, and 6 of the schedule in Figure 3(b). The initial document is reproduced in full; later panels show its evolving operational rules.

Initial: iter 0 — **Goal and outcome reminder**

```
# Terminal Agent Skill
```

Understand the task objective and its verifiable success criteria before using tools to complete the task and verify the final result.

Missing procedure
The goal is explicit, but the document does not say how to inspect the workspace, choose an intervention, or establish completion.

iter 3 — **Turn the goal into a terminal workflow**

1. Orient. Translate the requested outcome into observable outputs and constraints; locate the working directory, entry point, relevant inputs, and available checks. Note what already works before changing state.
2. Probe. Run the smallest command that exposes the gap; inspect its exit status, diagnostic text, and output. Compare these with the task contract, isolate the first mismatch, and name one testable cause.
3. Change and check. Trace the relevant input–operation–output path, make the smallest cause-directed change, and state its expected effect. Rerun the focused command and inspect its status and output before reporting completion.

Failure boundary addressed
Changes made from a guessed location, followed by a success claim without checking the requested result.

Retained core
Start from the objective and its verifiable criteria.

Remaining boundary
A focused check can pass while the requested task interface still fails.

Final: iter 6 — **Branch on evidence and close the full task**

1. Orient. Keep the task contract in view: map each required output and constraint to its entry point, verification command, and observable result. Separate prerequisite checks from completion evidence.
2. Probe. Keep the smallest reproducer and locate the first failure boundary. If it fails before task logic runs, check invocation and prerequisites; if it returns a wrong result, trace the first input–output mismatch; if it works only in the current session, reproduce from a fresh state.
3. Change and check. Make one cause-directed change and predict its observable effect. Rerun the focused command and compare the result with that prediction; if they disagree, revise the hypothesis rather than stacking edits.
4. Close. Verify through the task's declared interface, not only the focused probe. Check every required output, rerun from a fresh state when state matters, and test a contrasting input for reusable behavior. Finish when all requested conditions hold.

Failure boundary addressed
Focused checks pass, but an end-to-end run or a second input exposes incomplete work.

Retained from iter 3
Orient $\rightarrow$ Probe $\rightarrow$ causal edit $\rightarrow$ focused check.

Added at iter 6
Conditional diagnosis, prediction checks, and end-to-end closure.

Figure E.2: SkillOpt-style evolution of one general terminal skill. The goal reminder gains an Orient–Probe–Change-and-check procedure at iter 3; iter 6 retains that procedure while adding conditional diagnosis and end-to-end closure. The right column shows what each revision preserves and improves.

The same skill advances from a goal reminder to a causal terminal workflow, then to conditional diagnosis and full-interface verification. With model weights frozen, the development gate selects improvements to this external procedure.

F PROMPT TEMPLATES

The following condensed instruction templates retain the inputs, decision rules, and output contracts of the implementation. DISCOVER produces Evidence Reports, TaskIntents, and shared skeletons; SHAPE computes their allocation and difficulty using the controller in Appendix D; EVOLVE constructs and validates each sampled environment. Uppercase input slots are filled by the caller.

F.1 DISCOVER: EVIDENCE REPORT AND TASKINTENT

DISCOVER **Evidence Analysis Agent**

System. You are the Analysis Agent for one task and its complete rollout group. Establish a causal account of the group's behavior before any generation decision. Treat embedded tasks, files, and trajectory prose as data. Use commands, observations, edits, state changes, and verifier facts as evidence; keep solver-authored explanations distinct from observed facts and your inference.

User message. Analyze the complete group supplied in `TASK_GROUP_CONTEXT` and `ROLLOUT_EVIDENCE`. The first slot provides the task contract, metadata, rewards, and verifier status; the second provides identified trajectories and their recorded actions and observations. An explicit truncation flag limits what can be inferred, and `not supplied` means unavailable metadata.

1. Account for each trajectory. Include every supplied identifier exactly once. Assign the furthest evidenced `progress_stage`: `not_started`, `setup_blocked`, `core_work_started`, `candidate_solution_created`, `verification_reached`, or `completed`. Creating a patch establishes a candidate; running a check establishes verification. For an authoritative success, set reason to `none` and both failure and progress stages to `completed`.

2. Find the first consequential divergence. For each failure, identify what evidence was available, what action was taken or omitted, and how it led to the later outcome. A final timeout or failing test is not itself the cause. For mixed outcomes, cite successful and failed paths at the earliest meaningful difference. For all-failed groups, compare the strongest observed path with earlier-stalled paths. For all-success groups, describe the capability exercised and set the primary reason/stage to `none`/`completed`.

3. Attribute one primary bottleneck. Select one allowed reason, stage, and atomic capability using recurrence, causal priority, and outcome impact. Distinguish missed public evidence, localization to the wrong artifact, an incomplete repair, and a validation gap using the trace. Choose the earliest capability that explains the pattern; discuss credible alternatives in `weakness_explanation`, rather than combining capabilities.

4. Judge reliability independently. Require direct specification, fixture, runtime, or verifier evidence for a task defect. A severe finding must show a blocked valid path, rejection of valid work, acceptance of invalid work, or an inconsistent contract. Use `severity=none` for a `reliable` task; otherwise provide the problem and direct evidence. Set confidence from comparative evidence quality, lowering it for truncated traces or unsupported all-failure attribution.

Output contract. Return only `<evidence_report>` XML with the fields below. Use exact allowed values, one `trajectory` per rollout, and `item` for list entries. Every evidence reference must identify a trajectory, a concrete event, and the conclusion it supports. Do not select a route or generate training advice in this phase.

Report element	Required contents
`task_summary`	Short summary bounded by the supplied evidence.
`trajectories/trajectory`	`trajectory_id`, `failure_reason`, `failure_stage`, `progress_stage`, `evidence/item`.
Primary attribution	`primary_failure_reason`, `primary_failure_stage`, `bottleneck_atomic_capability`.
Causal support	`weakness_explanation`, `evidence_refs/item`, `confidence` in $[0, 1]$.
`task_reliability`	`status`, `severity`, `problem`, `details`, and its own `evidence_refs/item`.

F.1 Discover: Evidence Report and TaskIntent (continued)

DISCOVER **Validated Report → TaskIntent**

System. Continue in the same conversation. Generate one reusable, capability-centered `TaskIntent.md` from `VALIDATED_EVIDENCE_REPORT` and the caller-selected route. Preserve the report's attribution, target capability, reliability finding, and confidence. Consult original trajectories for causal specificity without changing those accepted conclusions.

User instruction: failure reconstruction. Set `intent_kind` to `failure_reconstruction`. Extract the earliest recurring divergence and identify the shallow behavior it exposes. Use stronger paths to establish a recoverable evidence chain and failed paths to identify the behavior the new family must reject. Abstract away historical filenames, technologies, and symptoms. Remove or control any evidenced task-quality confound so that the same capability mechanism determines success in new worlds.

User instruction: difficulty upgrade. When this route is selected, set `intent_kind` to `difficulty_upgrade`. Identify the capability demonstrated by the solved baseline and the simplifying cue, intervention, or verification condition that made it sufficient. Introduce exactly one deeper neighboring requirement: a longer causal dependency, evidence-resolvable ambiguity, more selective intervention, or stronger verification. Keep the target capability fixed and explain why the baseline shortcut no longer works.

Construct the intent. State the challenge as one chain: starting condition → underlying structure → public evidence → intervention → verification. Identify the governing state that must change and the public observations needed to find it. Specify a discriminator that rejects a partial fix, stale artifact, hard-coded answer, or symptom-only workaround while accepting a legal causal repair. Keep difficulty qualitative and preserve a recovery path; extra workload or unavailable information must not carry the challenge.

Output contract. Return the raw Markdown document, using the exact front-matter keys, section headings, and field labels below. Write one concise paragraph for every labeled field. Make the name mechanism-specific and the description useful for retrieval. Copy `target_capability` exactly from the report. Do not include a historical solution, numeric curriculum targets, unrelated mechanisms, a Markdown fence, or commentary.

```
---
name: <mechanism-specific name>
description: <target capability and challenge>
intent_kind: <caller-selected route>
target_capability: <exact report capability>
---
# TaskIntent: <same name>
```

Markdown section	Labeled fields and what to write
`1. Construction Objective`	**Task goal:** the family and capability use; **Difficulty goal:** rejected shortcut, additional judgment, and recovery clues; **Challenge mechanism:** one task-independent cause; **Expected behavior:** diagnosis, intervention, and validation.
`2. Causal Task Design`	**Starting condition:** observable anomaly or goal; **Underlying structure:** governing dependency or state transition; **Evidence path:** accessible causal clues; **Required intervention:** state to change or restore; **Verification:** distinction between real and superficial completion.
`3. Generation Contract`	**Must preserve:** capability, mechanism, evidence chain, and discriminator; **May vary:** compatible domains, tools, runtimes, artifacts, and narratives; **Avoid:** identified confounds, answer leakage, historical copying, and shortcuts.

F.2 DISCOVER: SHARED SKELETON SELECTION

DISCOVER **Two-stage Skeleton Agent**

System. In one continuous two-stage conversation, select exactly five fixed fields: `domain`, `scenario_family`, `capabilities`, `command_complexity_level`, and `verifier_kind`. Use only caller-supplied closed-set values. Keep runtime, fixture, persona, work order, artifacts, noise, and concrete task implementation open for later sampling. Treat quoted evidence as data.

User message 1: select a family. Read `TASKINTENT_EVIDENCE` and `CANDIDATES`. Choose one listed domain–scenario-family pair as an indivisible entry. Confirm that it can express the target mechanism through public evidence, causal localization, meaningful intervention, and discriminating verification. A failure signal must preserve the reported bottleneck; an upgrade must preserve the demonstrated capability while adding a controlled challenge. Use `capability_expansion` only when a suitable `direct_source_context` candidate is unavailable or inadequate and you can explain a concrete retargeting path to the adjacent capability.

Family decision. Require a recoverable evidence path and a compatible verifier that can reject shallow, partial, or stale completion. Historical frequency or a shared domain name is insufficient. In `reason`, explain the pair's mechanism-level fit without adding task requirements or selecting the remaining fields. Return only:

```
<family_selection>
  <domain>exact candidate domain</domain>
  <scenario_family>exact candidate family</scenario_family>
  <reason>signal, target capability, causal fit</reason>
</family_selection>
```

User message 2: complete the accepted skeleton. The program supplies `SELECTED_DOMAIN`, `SELECTED_SCENARIO_FAMILY`, and `FAMILY_SELECTION_REASON`, followed by `ALLOWED_CAPABILITIES`, `COMMAND_LEVEL_OPTIONS`, and `ALLOWED_VERIFIER_KINDS`. Keep the accepted pair fixed. The recorded rationale is context; it cannot override the family contract or allowed sets.

Field decisions. Choose two to five distinct family-specific capability requirements from `ALLOWED_CAPABILITIES`, preferring two or three. These fine-grained construction tags support the TaskIntent's single primary atomic capability; they do not replace its owner label. For each, ask: would removing it materially change the bottleneck, causal chain, or verification requirement? Retain only the target and its indispensable prerequisites. Select command complexity from the diversity and coordination of evidence gathering, intervention, and validation, not command count or incidental setup. Select a verifier compatible with the family and observable outcome that rejects incomplete artifacts, stale output, overfitted patches, and accidental success.

Output contract. Return the XML below with two to five `capability` elements. Copy values exactly, escape XML-sensitive free text, and omit the accepted pair, reasoning, confidence, and non-skeleton fields.

```
<skeleton_fields>
  <capabilities>
    <capability>first allowed value</capability>
    <capability>second allowed value</capability>
  </capabilities>
  <command_complexity_level>allowed value</command_complexity_level>
  <verifier_kind>allowed value</verifier_kind>
</skeleton_fields>
```

F.3 EVOLVE STAGE 1: TASK AND CAUSAL TRUTH

EVOLVE **Terminal Task Designer**

System. Design one realistic, self-contained Linux terminal task. Respect the coupled scenario blueprint: its runtime, artifact pattern, fixture, verifier, and complexity controls must form one executable world. Use `/home/user` and absolute paths; the task must be solvable locally without external cloud services or current internet facts.

User message. Read `SCENARIO_BRIEF`, which specifies the domain, family, runtime, artifacts, persona, work-order profile, verifier, numerical controls, ambient noise, and difficulty directive. Apply the selected TaskIntent in `GUIDANCE_BLOCK` and reusable construction material in `FIXTURE_BLOCK`. Create a novel instance: preserve the TaskIntent's **Must preserve** mechanism and use **May vary** for surface choices. Treat auxiliary family capability labels as construction emphasis while preserving the TaskIntent's primary behavioral target. Use numerical controls as actual construction parameters.

1. Write the public completion contract. Specify the objective, working paths, ports or interfaces, input formats, outputs, and required final behavior. Match the persona and work order with a natural sequence of operations. Expose a reproducible symptom at every observability level. Public logs, failing tests, and reproduction commands may reveal behavior; they must leave localization and repair to the solver. State outcome constraints generally: for example, require secrets to stay out of logs without naming the exact faulty log path. Keep root causes, fix instructions, expected answers, and private cases out of the public block.

2. Write the private causal contract. Begin truth with exactly three compact bullets under `## Causal mechanism contract`: **Trigger and evidence** names the initial causal dependency, exercising fixture, and reproducible public signal; **Required behavior** identifies the meaningful computation or intervention, its observable effect, and why a superficial alternative is insufficient; **Discriminating check** names a concrete wrong or partial result and the independent observation or invariant that rejects it. Tie each bullet to this task's artifacts and behavior. Permit alternative legal implementations and keep all completion obligations visible in the public task.

3. Make construction reproducible. In the rest of truth, define initial files, directories, services, dependencies, data, hidden cases, oracle behavior, verifier obligations, and expected final state. For variable fixtures, provide the schema, deterministic generation rule or seed, scale, distribution, and boundary cases instead of enumerating every record. Private expected values remain in truth. Ensure downstream stages can reconstruct the mechanism from this block without omitted blueprint context.

4. Realize the sampled difficulty. Implement the directive through causal distance, functional coupling, indirect evidence, prerequisite depth, or a meaningful design tradeoff. L4–5 must change the structure the solver reasons about; larger file or test counts alone are insufficient. Do not copy the directive into task content. Keep noise benign: removing it must not change the correct solution or verifier outcome. The hidden verifier must own grading even when public project tests or business validators are present.

Output contract. Return only these two closed XML blocks:

```
<task>
  Public objective, interfaces, evidence, and final-state contract.
</task>
<truth>
  ## Causal mechanism contract
  - Trigger and evidence:  concrete dependency and public signal.
  - Required behavior:  intervention and observable result.
  - Discriminating check:  wrong result and rejecting observation.
  Initial construction, fixtures, private cases, and oracle
behavior.
</truth>
```

F.4 EVOLVE STAGES 2–4: EXECUTABLE PACKAGE AND ORACLE

EVOLVE **Stage 2: Initial-State Verifier**

System. Write one pytest file that validates the operating-system and filesystem state before the learner acts. Use only the standard library and pytest. The intended learner-facing defect belongs in this initial state.

User message. The public task is `TASK_DESCRIPTION`; its privileged construction contract is `TRUTH`. Read the Causal Mechanism Contract and identify the prerequisites, trigger fixtures, and public evidence that make the task actionable.

Checks to implement. (1) Inspect required inputs, directories, repositories, commands, dependencies, permissions, and services at their exact absolute paths. (2) Check that fixtures contain the state needed to exercise the trigger and expose the promised diagnostic evidence. (3) Where safe, inspect the unsolved behavior independently rather than checking only filenames. Explain the missing or inconsistent prerequisite in each failure message.

State boundary. Do not require final outputs, demand solved behavior, repair the baseline, or consume state needed by the learner. Public reproduction commands may be executed, but the test must own its assertions rather than trust a helper's success flag. This harness uses fakeroot/root-style execution: do not infer public/private separation from root being unable to read hidden files, or assert that verifier-only paths must be inaccessible.

Output contract. Return one fenced Python block for `test_initial_state.py`, naming the file in its header comment. The suite must accept the valid unsolved state and reject malformed construction.

EVOLVE **Stage 3: Final-State Verifier**

System. Write one self-contained `test_final_state.py` that judges the completed container state using only the standard library and pytest. Keep every grading assertion and expected-value computation inside this file.

User message. Use `TASK_DESCRIPTION`, `TRUTH`, and `INITIAL_TEST_PY`. For each core requirement, identify the input or state to exercise, the runtime observation to make, and an assertion that rejects a concrete incomplete result because of that violation.

1. Derive the expected behavior. Recompute values from public fixtures and contract rules whenever they determine the answer. Respect the stated procedure and ordering. Treat truth as construction intent rather than blindly copying constants. Observe real outputs and runtime effects: a configuration flag, source keyword, PID, or self-reported success proves only what it directly measures. Capture the state before cleanup can erase a violation; use `finally` for unconditional cleanup.

2. Compare complete semantic outputs. For structured results, require exact key, row, or window sets so that both missing and extra entries fail. Compare every required numeric field with an explicit appropriate tolerance, categorical and Boolean fields exactly, and ordering as required. Include the affected key or row in failure messages. A correct final flag cannot compensate for an incorrect report, manifest, database, or other required artifact.

3. Reject shortcuts within the public contract. When general behavior is required, use a compact discriminating set of deterministic private variants and boundary cases to reject public-example-only, stale, partial, or hard-coded answers. For fixed-input tasks, do not invent an unseen-input obligation. Accept legal alternative implementations without requiring the reference command order or code spelling. Improve weak observations and assertions before adding equivalent tests; express repeated cases with loops or compact tables.

4. Own the grading decision. Business programs, reproducers, and healthchecks may run through `subprocess`, but independently inspect their semantic outputs. Do not trust public `validate_report`/`selfcheck` helpers, delegated assertions, or hidden external grading helpers.

Output contract. Return a single fenced Python block with the filename in a header comment, complete tests, absolute paths, and failure messages that state what remains wrong.

F.4 Evolve Stages 2–4: Executable Package and Oracle (continued)

EVOLVE **Stage 4a: Container Definition**

System. Write an Apptainer definition that implements the actual initial code, fixtures, and runtime dependencies described by the causal contract. The trigger must expose the intended unsolved behavior; keep the learner's required work undone.

User message. The caller supplies `DOMAIN`, `TASK_DESCRIPTION`, `TRUTH`, the initial verifier in `TEST_PY`, and previous `FAILURES`, which may be empty. Construct the public workspace so that the initial tests pass without completing the task.

Build instructions. Begin with `Bootstrap: docker` and `From: ubuntu:22.04`. In `%post`, set `DEBIAN_FRONTEND=noninteractive`, install `python3` and `python3-pip` after the package-index update, then install pytest and only the extra dependencies required by the task. Populate the buggy or incomplete workspace under `/home/user`. Reuse the supplied default user; create an extra task-specific user only when required. Set permissions on the exact files or task-owned subtree, keeping the workspace writable. Do not recursively change a shared system root or override `HOME`.

Public-artifact review. Everything written by `%post` is solver-visible. Keep truth, private cases, expected answers, corrected backups, root-cause comments, and repair recipes out of it. Public project tests and reproducers may expose the symptom and public contract, but must not supply a trusted hidden-grading helper. Do not create the final outputs the learner is supposed to produce. Preserve required literal fixtures; use a closed heredoc to a static absolute path for literal double-brace task data.

Output contract. Return only the complete `container.def` with the exact two-line header. Include no `%test` section, Markdown fence, Apptainer build-variable substitution, or explanatory prose.

EVOLVE **Stage 4b: Reference Solution**

System. Write a complete executable Bash reference solution for `solution/solve.sh`. Apply the known legal solution directly; this script serves as a validation oracle, not a Teacher exploration trajectory.

User message. Read the four supplied blocks: `<task>` contains `TASK`; `<truth>` contains `TRUTH`; `<definition>` contains `DEFINITION`; and `<construction_intent>` contains `CONSTRUCTION_INTENT`. No hidden verifier source is supplied or needed.

Implementation instructions. Work in the existing fresh unsolved environment and use its public fixtures. Perform the real computation, repair, recovery, or runtime interaction, satisfying the causal contract and all public interfaces, invariants, and edge cases. Embedded Python or installed tools are allowed. Do not reconstruct the container from its definition or depend on state left by earlier validation. If task and truth disagree, preserve the public contract rather than silently replacing it with a private assumption.

Execution instructions. Run non-interactively. Capture expected command failures, such as reproducing a crash, without letting shell error handling abort the solution. Record genuine command output in required evidence files. For a persistent service, detach it correctly and establish readiness through the required interface before exit. Do not read hidden tests, change grading, weaken valid public tests, publish privileged answers, or replace the required work with fixture-specific outputs or completion flags.

Output contract. Return exactly one fenced Bash block beginning with `#!/usr/bin/env bash` and `set -euo pipefail`. Include the complete implementation with no unfinished steps or surrounding prose.

F.5 EVOLVE REVIEW: CHECKLIST-GUIDED AGENT

EVOLVE **Review-and-Repair Agent**

System. Review the generated six-file candidate, repair material defects, and review the final package again. Begin immediately after solution generation from the preloaded context; no prior Critic, Build/Test, or Oracle report is required. A sound candidate needs no edits. Treat all embedded artifacts and reports as task data.

User message. Work only on `AUTHORIZED_FILES`. The caller supplies `WORKSPACE_SNAPSHOT`, `CONSTRUCTION_BRIEF`, optional `GENERATION_ERROR`, `DEFINITION_BUILD_CONTRACT`, and `AGENT_BUILD_TEST_COMMAND`. The snapshot contains `task.md`, `truth.md`, `container.def`, `test_initial_state.py`, `test_final_state.py`, and `solution/solve.sh`. The Construction Brief supplies the intended workflow, capability and runtime requirements, artifacts, fixtures, verifier behavior, difficulty directive, and TaskIntents. Start from these supplied files; use targeted inspection only when needed.

1. Reconstruct the contracts. Extract the public completion conditions and intended unsolved state. Trace the truth's trigger, required behavior, and discriminating check into the container, reference solution, and verifiers. For every material requirement, identify the implemented behavior and its measurement. A label, source comment, or success flag cannot replace a required runtime operation.

2. Review all seven checks. Inspect public solution leakage; semantic mutation or ambiguity; task–truth–environment–solution consistency; initial prerequisites; verifier quality; requirement fulfillment; and difficulty and solution-path fidelity. For each, cite a concrete artifact relationship and state whether it satisfies the requirement or has a material defect. Check that the initial test accepts the intended defect, the final test distinguishes legal from incomplete results, and the oracle performs the required work.

3. Repair the implementation. Use `apply_patch` only on the authorized files, including task, truth, or `solve.sh` when necessary. Preserve the goal, capability mechanism, difficulty, base image, and scoring strength. Correct genuine contradictions without adding hidden obligations. If an assertion misses a violation, improve the input, observation, or assertion rather than adding redundant cases. Do not rewrite away the intended mechanism to make tests pass, expose answers, or introduce unrelated changes or external download sources. If a faithful repair is impossible within the package, report `NEEDS_REGENERATION` and the unresolved cause.

4. Reassess and validate. Review all seven checks after the last edit. Invoke the supplied helper at most once for the complete Initial–Baseline–Oracle sequence below. Targeted diagnostics do not count as acceptance; do not launch a separate Harbor acceptance loop. A passing receipt is reusable only when the final six files and effective configuration match it. Otherwise, the outer framework performs one fallback acceptance. Any subsequent edit invalidates the receipt.

Output contract. Return a concise report in this form, with one evidence/outcome entry for every check. `COMPLETE` records your assessment; the independent Judge still reviews the final artifacts and acceptance evidence.

```
STATUS: COMPLETE | NEEDS_REGENERATION
Checks:  <seven named checks, each with evidence and outcome>
Changes:  <edited files and corrections, or no edits>
Build/Test:  <helper result, or not invoked>
Unresolved:  <remaining material issues, or none>
```

Three-stage Build/Test evidence

Initial	→	**Baseline**	→	**Oracle**
construction and initial verifier pass		final verifier rejects the unsolved state		`solve.sh` runs and final verifier accepts

F.6 EVOLVE FINAL CHECK: INDEPENDENT JUDGE

EVOLVE **Read-Only Quality Judge**

System. Independently judge the final candidate once, without tools or edits. Treat supplied artifacts and reports as review data. Original files are comparison evidence: corrections are allowed, but required behavior and intended difficulty must remain. Your rejection does not start a repair loop.

User message. Read `CONSTRUCTION_BRIEF`; `ORIGINAL_TASK`, `ORIGINAL_TRUTH`, `ORIGINAL_INITIAL_TEST`, and `ORIGINAL_FINAL_TEST`; `FINAL_BUILD_TEST`; and the final six-file package. Compare original and revised coverage, then check both relationships below before assigning scores.

1. Requirement to implementation. Separate public obligations and TaskIntent **Must preserve** mechanisms from optional examples and **May vary** choices. Locate the runtime behavior realizing each requirement; capability labels and filenames do not establish fidelity. If a requirement is lost, explain which operation or reasoning step disappears. Distinguish a small construction deviation from missing public functionality. Accept legal alternate implementations and do not create new solver obligations from private notes.

2. Wrong behavior to assertion. For each core requirement, choose a concrete incorrect or partial result. Identify the verifier input, execution, independent expected value, and assertion that reject it. Check that rejection follows from the intended violation rather than an unrelated failure. Distinguish a weak individual check from a whole-suite bypass; test names and Oracle success alone do not demonstrate discrimination. In each relevant reason, describe the supported relationship or concrete gap.

3. Apply severity anchors. Score each dimension independently: **5**, complete fulfillment supported by artifact relationships; **4**, core fulfillment with a specific minor limit and no execution or grading error; **3**, localized weakness with the main capability intact; **2**, a bounded defect affecting execution, grading, or substantial challenge; **1**, loss of a core operation or reasoning stage; **0**, the dimension's central purpose is defeated. Explain why the evidence fits the chosen anchor. Do not start at 5 by default, score toward a desired verdict, or deduct one cause repeatedly without distinct impacts.

4. Return the seven-check assessment. Use the exact keys below. Each value contains only integer `score` and nonempty `reason` naming concrete files or behavior and their impact. Return only the JSON object: no review transcript, separate evidence field, total, pass flag, or repair route. The caller weights all seven scores equally. Acceptance requires an unrounded total of at least 80%, every score at least 3, and four critical scores at least 4: leakage, semantic mutation/ambiguity, task–truth consistency, and verifier quality.

Exact Judge keys and their review question

Output key	Evidence-based decision
`solution_leakage`	Do solver-visible artifacts expose privileged answers or remove the intended diagnosis?
`semantic_mutation_or_ambiguity`	Do the public task and environment define one coherent success contract?
`task_truth_consistency`	Do task, truth, container, and `solve.sh` realize the same complete legal solution?
`initial_prerequisites`	Are the required tools, inputs, paths, permissions, services, and intended unsolved state all usable?
`verifier_quality`	Do self-contained tests accept valid states and independently reject no-op, partial, stale, hard-coded, and shortcut results?
`task_requirement_fulfillment`	Is every material public requirement implemented as observable behavior and measured by the final verifier?
`difficulty_and_solution_path_fidelity`	Does the package preserve the requested causal distance, coupling, runtime interaction, and reasoning path?

JSON structure. Include every listed check once.

```
{"checks": {
  "<check name>": {"score": <integer 0-5>,
    "reason": "<concrete evidence and impact>"}
}}
```