SPADE ♠: SELF-PLAY IN ADAPTIVE SYNTHETIC EXECUTABLE ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Continuous self-improvement requires an ever-expanding pool of self-generated, diverse, adaptive goals. For language agents, existing training environment pools (hand-curated, statically synthesized, or frozen-verifier) keep the goal distribution fixed as the learner scales. We introduce `SPADE` (Self-Play in Adaptive Synthetic Executable Environments), a self-play RL framework in which a single LLM plays two roles: an `Environment Designer` that writes complete, long-horizon training environments as executable code with an OpenAI Gym-style `reset()`/`step()` interface, and a `Reasoning Agent` that learns to act in them. Each is a stateful, multi-turn environment (state transitions, reward functions, and verification code), so one interface spans reasoning problems and multi-step agentic tool use. The `Reasoning Agent`'s regret is estimated using the gap between its reward with and without privileged hints. By optimizing this regret signal the `Environment Designer` learns to target environments at the edge of the agent's capabilities while keeping them feasible. Through extensive experimentation, we find several components critical to success: grounding the `Environment Designer` on documents sampled from a large pretraining corpus, and giving it an accumulated environment memory. Scaling to 30B-parameter models, `SPADE` improves over the strongest fixed-environment baseline by $+5.3$ on average across eight held-out reasoning benchmarks, and lifts the tool-use setting by $+5.7$ on BFCL v4 multi-turn and $+13.9$ on ACEBench-Agent. On terminal settings, `SPADE` improves Terminal-Bench 2.1 by $+7.2$ points and SWE-bench Verified by $+8.0$ over the base model. By making environment design itself a learnable component, `SPADE` takes a concrete step toward open-ended self-improvement.[1]

1 INTRODUCTION

Agentic AI has become broadly capable: language models now reason over long horizons, use tools, search the web, and operate computers (OpenAI, 2024; DeepSeek-AI, 2025; Wang et al., 2025c; Zhang et al., 2025c). These gains increasingly come not from pretraining alone but from learning through experience, where a model improves from its own trajectories of interacting with an environment and the rewards they return (Silver & Sutton, 2025). As high-quality human text is a finite resource (Villalobos et al., 2024), the bottleneck for further agentic progress is increasingly the supply of training environments: interactive tasks with verifiable rewards. Building them has become a central industry investment, with major labs weighing environment budgets exceeding $1 billion a year and a new class of startups raising nine figures to supply them (Zeff, 2025; PYMNTS, 2026). Yet whether hand-built or synthesized, these environments form a fixed pool that does not adapt as the learner improves, so an agent stops improving once it exhausts them.

Several lines of work try to keep agents improving, each with its own limitation. *Harness engineering* improves how an agent acts within a given environment at inference time, through tool orchestration and self-correction (Yao et al., 2023; Shinn et al., 2023), but it changes no weights, so its gains are tied to each hand-built harness, rather than improving general capabilities the model carries to new tasks. *Human-curated scaling* (Zhang et al., 2025a; Guertler et al., 2025; Liu et al., 2025b) builds diverse environments to train across, but scales only as fast as people can write them. *Synthetic generation* (Wang et al., 2026b; Tu et al., 2026; Zhu et al., 2026; Gandhi et al., 2026; Zeng et al.,

[1]The code is available at `https://anonymous.4open.science/r/spade-iclr2027`

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.

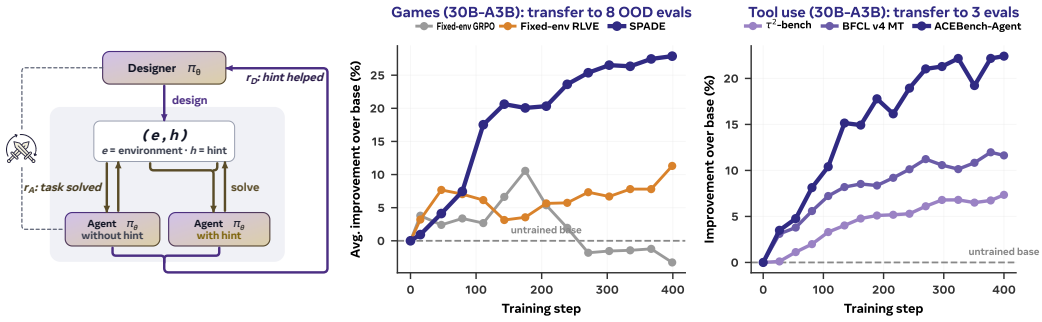


Figure 1: **SPADE designs and solves its own training environments. Left:** shared-model self-play with hint-based regret. **Middle:** average relative gain across eight games-setting benchmarks. **Right:** per-benchmark relative gains on three tool-use evaluations. Both plots use Qwen3-30B-A3B.

2025) produces environments programmatically, but the generators are fixed, so the environment space does not grow with the agent (Song et al., 2024). *Self-play* (Zhao et al., 2025; Huang et al., 2026b; Liu et al., 2026a) lets a model improve through dual roles, but ungrounded self-play is bounded by information symmetry, unable to pose challenges beyond its own knowledge and prone to amplifying its errors (Chae et al., 2025); corpus grounding mitigates this (Liu et al., 2025a), yet most methods still generate *tasks* (a problem with a sparse terminal reward) rather than complete *multi-turn environments* (state transitions, reward functions, and verification code). None produces a self-improving system whose environments adapt as the agent improves.

We introduce `SPADE` (Self-Play in Adaptive Synthetic Executable Environments), a framework where a single LLM plays two roles: an `Environment Designer` that produces complete training environments as executable Python code, and a `Reasoning Agent` that learns from them. Each environment implements a Gym-style interface (the standard `reset()`/`step()` API) (Brockman et al., 2016), representing a full Markov decision process (MDP) with state transitions and reward functions. This *code-as-environment* representation unifies single-turn settings (one step to terminal reward) and multi-turn agentic tasks (sequential interaction with transition dynamics) under a single interface. Because any computable MDP can be written as a program, the `Environment Designer` can express any such environment in code, rather than only those a hand-designed parameterization allows. Unlike prior works where the environment generator is frozen or trained on proxy signals, `SPADE`'s `Environment Designer` is itself trained via RL with a *hint-based regret* signal that targets solvable environments precisely at the frontier of the `Reasoning Agent`'s capability, creating co-evolution where the environment distribution shifts as the `Reasoning Agent` improves (Figure 2).

Overall, our work makes the following contributions:

1. We introduce a *general framework for co-evolving environment synthesis and agentic capabilities through self-play*, where the same LLM both generates executable environments as code and learns to solve them. By representing environments as Python programs with a Gym-style interface, the framework turns environment design into a learnable, RL-trained component of post-training, a step toward continual open-ended self-improvement.
2. We introduce a *hint-based regret reward* for the `Environment Designer`, which rewards the gap between `Reasoning Agent` performance with and without privileged hints, a lightweight estimate of minimax regret (Dennis et al., 2020) that targets environments at the frontier of the `Reasoning Agent`'s capability. Unlike a pure adversary or a cooperative designer, hint-based regret yields constrained competitive dynamics that reward environments which are both solvable and at the learning frontier, with an equilibrium analysis in Appendix D.
3. We design a *corpus-grounded and memory-augmented design pipeline* spanning cognitive-skill games and tool-use tasks. While prior environment-generation work targets a single domain, `SPADE` demonstrates improvements in both settings, with environments grounded in corpus knowledge and accumulated experience.

Across three Qwen3 backbones, `SPADE` outperforms fixed-environment baselines in games; at 30B-A3B it leads by $+5.3$ points and improves BFCL v4 multi-turn by $+5.7$ and ACEBench-Agent by

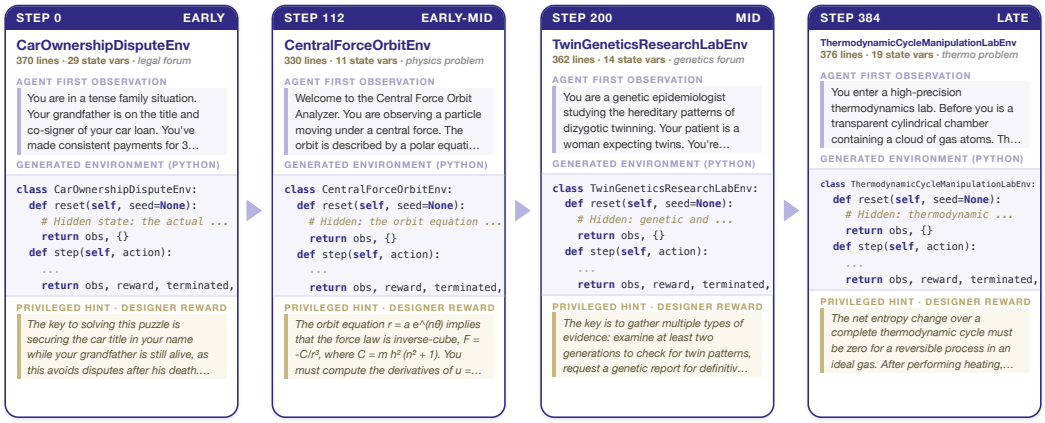


Figure 2: **SPADE generates an adaptive, multi-turn curriculum.** Four environments the `Environment Designer` produces over one 30B-A3B run, steps 0 to 384; each card shows the `Reasoning Agent`'s first observation, the generated Python environment (a complete MDP with a `reset()`/`step()` interface), and the designer-written hint. Tasks shift toward state-gated interaction as the `Reasoning Agent` improves.

+13.9. The approach also extends to terminal tasks: on a Nemotron backbone, SPADE improves over base by +7.2 points on Terminal-Bench 2.1 and +8.0 on SWE-bench Verified. Ablations favor hint-based regret and the full adaptive, corpus-grounded, memory-augmented configuration. Qualitative analysis shows that corpus grounding keeps the generated environments diverse. A single model thus improves not only how it reasons and acts, but the worlds it builds to learn in, a concrete step toward open-ended, continual self-improvement.

2 Related Work

For an extended background and comparison to the literature see Appendix B, summarized here:

Self-Play for LLMs Self-play carried game AI from TD-Gammon (Tesauro, 1995) to AlphaZero (Silver et al., 2017; 2018) and beyond (Vinyals et al., 2019; Meta Fundamental AI Research Diplomacy Team (FAIR)† et al., 2022); asymmetric self-play (Sukhbaatar et al., 2017) split it into a proposer that sets challenges and a solver that meets them. LLM self-play first relied on curated seeds (Chen et al., 2024; Yuan et al., 2024; Cheng et al., 2024). SPIRAL (Liu et al., 2026a) trains a single model by competitive self-play on fixed zero-sum games, while AZR (Zhao et al., 2025), SQLM (Chen et al., 2025c), R-Zero (Huang et al., 2026b), and SPICE (Liu et al., 2025a) have the model generate its own tasks from minimal seeds with proxy rewards aimed at the learnable frontier. These task-generating methods propose one problem with a terminal reward, and can collapse when model capacity bounds the reachable solutions (Chae et al., 2025) or the proxy reward admits degenerate outputs (Shafayat et al., 2025). SPADE instead generates *full multi-turn MDPs* as executable code, with the verifier inside each environment, and trains the `Environment Designer` on regret measured from `Reasoning Agent` return.

Unsupervised Environment Design UED (Dennis et al., 2020) trains an agent on environments chosen by an adversary; PAIRED pays the adversary minimax regret, so it selects environments that are hard yet solvable, and later work improved level curation (Jiang et al., 2021b;a; Parker-Holder et al., 2022), stability (Mediratta et al., 2023), and regret estimation (Rutherford et al., 2024). POET (Wang et al., 2019) co-evolves environments and agents, and OMNI-EPIC (Faldor et al., 2024) has a foundation model write environments as code under a frozen interestingness judge. These systems search a parameterized design space or keep the generator frozen. SPADE keeps the regret objective and makes the designer itself the learner: it writes MDPs as programs, is trained by RL, and co-evolves with the `Reasoning Agent` online.

Environment Synthesis and Scaling The move from single-turn RLVR (DeepSeek-AI, 2025) to multi-turn agentic RL (Wang et al., 2025c; Zhang et al., 2025c) made the environment pool the bottleneck. Recent systems synthesize agentic worlds and toolsets (Wang et al., 2026b; Tu et al.,

2026; Dong et al., 2026), terminal and computer-use sandboxes (Zhu et al., 2026; Gandhi et al., 2026; Xue et al., 2026a), and large pools of verifiable environments (Zeng et al., 2025; Cen et al., 2026; Zhang et al., 2025a), with curriculum selection over fixed pools for difficulty adaptation (Chen et al., 2025d; Xu et al., 2026a). Among these the generator is frozen, hand-engineered, or trained on separate signals; SPADE trains it in the same loop as the Reasoning Agent, on the Reasoning Agent's own returns.

3 PRELIMINARIES

We review two components of SPADE: Gym-style MDPs for representing environments, and minimax-regret environment design for training the two roles with GRPO.

Markov Decision Processes and Gym-Style Environments. An MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, R, \rho_0)$ of states, actions, transition and reward functions, and an initial-state distribution. In the Gym/Gymnasium interface (Brockman et al., 2016; Towers et al., 2025), `reset()` samples $s_0 \sim \rho_0$, and `step`(a) applies T, returning the next observation, reward, and termination/truncation flags. The program state is the MDP state; observations may reveal only part of it. Appendix G.3.1 (Listing 1) gives a complete example.

Environment Design as a Minimax-Regret Game. Let π_θ be an LLM with parameters θ and $\mathcal{E}$ the set of valid executable environments. For a Reasoning Agent policy π and an environment $e \in \mathcal{E}$, let $R_\pi(e)$ be its expected verifier return and $R^\star(e) := \max_{\pi'} R_{\pi'}(e)$ the best return attainable on e. Unsupervised environment design (UED) (Dennis et al., 2020) trains the agent on environments chosen by an adversary that is paid the agent's *regret* $R^\star(e) - R_\pi(e)$: regret is zero on environments the agent already solves and on environments no policy can solve, and positive only in between, so the adversary is drawn to solvable environments the agent still fails. With the Environment Designer choosing a distribution $D \in \Delta(\mathcal{E})$ over environments and the Reasoning Agent choosing π, SPADE plays the resulting minimax game,

$$\min_{\pi} \max_{D \in \Delta(\mathcal{E})} \mathbb{E}_{e \sim D}\big[R^\star(e) - R_\pi(e)\big]. \tag{1}$$

$R^\star(e)$ is not observable and must be estimated: PAIRED (Dennis et al., 2020) trains a separate antagonist, while SPADE uses the return of the same policy given a privileged hint (Section 4.2), which equals $R^\star(e)$ under the articulated-hint assumption of Appendix D. Both roles are π_θ under different role prompts, and each role's payoff is optimized with GRPO (Shao et al., 2024) (DPPO for terminal tasks); the objective and its per-role advantage normalization appear in Appendix E.1.

4 SPADE: SELF-PLAY IN ADAPTIVE SYNTHETIC EXECUTABLE ENVS

SPADE is an end-to-end self-play framework where a single LLM, π_θ, alternates between generating executable training environments and learning to solve them (Figure 3; Algorithm 1). Its three components create an adaptive curriculum: § 4.1 details how dual-role self-play couples environment generation with Reasoning Agent (RA) learning; § 4.2 explains how hint-based regret steers the Environment Designer (ED) toward environments at the Reasoning Agent's learning frontier; and § 4.3 outlines the pipeline that grounds and validates these executable environments.

4.1 DUAL-ROLE SELF-PLAY AND CODE-AS-ENVIRONMENT

SPADE formulates adaptive environment self-play as a game where a single model π_θ acts in two roles via role-specific system prompts: designing environments (role=D) or solving them (role=A). Let $\mathcal{E}$ denote the space of all valid environments. In the Environment Designer role (π_D), π_θ produces an executable environment $e \in \mathcal{E}$ as a Python program implementing the Gym-style `reset()`/`step()` API (Section 3; a minimal example is shown in Appendix G.3.1), where the transition function $T(s' \mid s, a)$ and reward function $R(s, a)$ are both encoded in the `step()` implementation. The Environment Designer also emits a privileged hint for each environment. A hint h is task-relevant information attached to an environment (for example, a partial solution sketch or a key structural observation); revealing h makes the environment easier to solve.

In the Reasoning Agent role (π_A), π_θ interacts with e via sequential actions, receiving observations and rewards. This *code-as-environment* representation unifies single-turn settings (`reset()` $\rightarrow$ `step(answer)` $\rightarrow$ terminal reward) and multi-turn agentic settings (`reset()` $\rightarrow$ `step()`

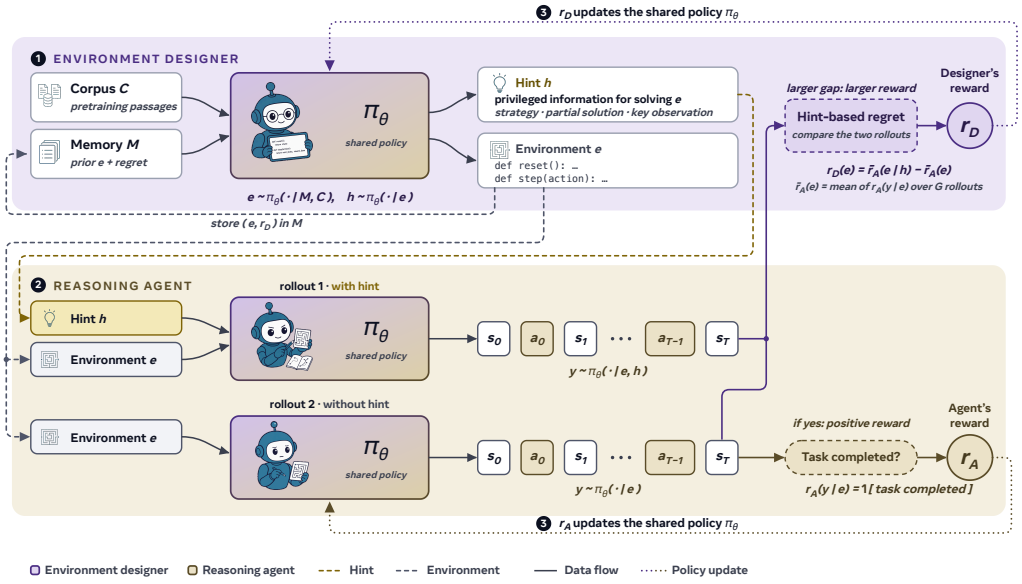


Figure 3: **The SPADE framework.** Top: the `Environment Designer` uses environment memory M and pretraining corpus C to generate environment e and hint h. Bottom: the `Reasoning Agent`'s return gap with versus without h rewards the `Environment Designer` (Eq. 2); task correctness rewards the `Reasoning Agent`. Both roles update π_θ with GRPO.

$\rightarrow \cdots \rightarrow$ `done`) under a single interface and training pipeline. This representation provides the vast task space that AI-GAs (Clune, 2019) argue is necessary for open-ended self-improvement: any computable MDP can be expressed as a Python program, so the `Environment Designer` can express any such environment in code rather than being confined to a hand-designed environment parameterization. Both roles share parameters θ, so updates for either role affect the same policy. Each self-play cycle first collects `Environment Designer` trajectories, and then `Reasoning Agent` trajectories from those generated environments. The `Environment Designer` receives hint-based regret (Section 4.2), whereas the `Reasoning Agent` receives task correctness from the environment's reward function. Each candidate environment is validated for syntactic correctness and executability before entering the training pool; validation details are in Appendix E.2.

Stabilizing joint two-role training. Optimizing one policy for two coupled objectives needs several stabilizers (Liu et al., 2026b). Following SPIRAL (Liu et al., 2026a), we normalize advantages per role, standardizing `Reasoning Agent` returns within each environment and mean-centering `Environment Designer` rewards within each skill, and upweight the rarer `Environment Designer` trajectories. We delay the `Environment Designer` update by k rollouts so the difficulty anchor can score each environment's `Reasoning Agent` win rate over the full window, and correct the off-policy with TIS (Yao et al., 2025). Further training details are in Appendix E.2.

4.2 Hint-Based Regret Reward

The `Environment Designer`'s reward for producing environment e is:

$$r_D(e) = \bar{r}_A(e \mid h) - \bar{r}_A(e), \tag{2}$$

where $\bar{r}_A(e \mid h) = \frac{1}{G}\sum_{i=1}^{G} r_A(y_i' \mid e, h)$ is the `Reasoning Agent`'s average return over G fresh rollouts y_i' sampled with the privileged hint h in context, and $\bar{r}_A(e)$ is the corresponding average over G rollouts without it. Three regimes emerge: high regret indicates an environment at the learning frontier (solvable with hints but not without); low regret where the `Reasoning Agent` succeeds with and without the hint indicates mastery; and low regret with low returns even with the hint indicates an intractable environment. This implements the minimax regret objective of PAIRED (Dennis et al., 2020) without a separate antagonist: the hint-equipped `Reasoning Agent` serves as the upper-bound policy. For a policy that best-responds to the hint, extra information cannot reduce expected return, so regret is non-negative at the optimum; for the current policy, expected regret can dip below zero when hints mislead it, which we observe for the smaller backbones (Figure 9).

Algorithm 1 SPADE ♠: Self-Play in Adaptive Synthetic Executable Environments

Require: Pretrained LLM π_θ; domain prompts $\{p_d\}$; batch size B; group size G; iterations N

for $n \leftarrow 1$ to N **do**
  `Environment Designer` role (role=D): ▷ generate environments
  for $b \leftarrow 1$ to B **do**
    Sample domain prompt p_d; generate $e_b \sim \pi_\theta(\cdot \mid p_d, \text{role}{=}D)$
    Validate e_b: syntax, executability; discard invalid
    Generate hint $h_b \sim \pi_\theta(\cdot \mid e_b, \text{role}{=}D)$ ▷ privileged info
  end for
  Shared `Reasoning Agent` rollouts (role=A): ▷ trained on *and* scored for regret
  for each valid environment e_b **do**
    $\{y_i\}_{i=1}^G \sim \pi_\theta(\cdot \mid e_b, \text{role}{=}A)$; $r_A(y_i \mid e_b) \in [-1, 1]$ ▷ no-hint plays
    $\{y'_i\}_{i=1}^G \sim \pi_\theta(\cdot \mid e_b, h_b, \text{role}{=}A)$; $\bar{r}_A(e_b \mid h_b) \leftarrow \frac{1}{G}\sum_{i=1}^G r_A(y'_i \mid e_b, h_b)$ ▷ hint plays
  end for
  Rewards and joint GRPO update (for each valid e_b and its rollouts):
  $r_D(e_b) \leftarrow \bar{r}_A(e_b \mid h_b) - \frac{1}{G}\sum_{i=1}^G r_A(y_i \mid e_b)$ ▷ $\max(0, \cdot)$ in training
  $\hat{A}^b_D \leftarrow r_D(e_b) - \text{mean}_{\text{same skill}}\{r_D(e_c)\}$; $\hat{A}^i_A \leftarrow \frac{r_A(y_i|e_b) - \text{mean}(\{r_A(y_j|e_b)\}_{j=1}^G)}{\text{std}(\{r_A(y_j|e_b)\}_{j=1}^G)}$ ▷ per-role advantages
  Update π_θ by clipped policy gradient on $\{\hat{A}^b_D\} \cup \{\hat{A}^i_A\}$
end for
return trained model π_θ

We formalize this intuition and show that, under idealized assumptions, every pure Nash equilibrium yields hint-free optimality on every valid environment (Appendix D).

Hint generation. For each environment, the `Environment Designer` is prompted to write a short hint giving the key insight or strategy and expected answer format, without revealing the exact answer. Appendix G.1.1 shows two high-regret tasks, hints and corresponding rollouts.

4.3 Environment Design

Corpus grounding. A generator conditioned only on its own output has no source of novelty outside its own weights, so it narrows toward the patterns it already favors and mode-collapses, the "invisible leash" (Chae et al., 2025; Zhang et al., 2026c; Jiang et al., 2025a). We treat an external corpus as the mechanism that lengthens that leash: *every* round the `Environment Designer` conditions on freshly sampled human corpus. In the games setting these are 10k mathematics and 5k science documents drawn from DCLM (Li et al., 2024) and MegaScience (Fan et al., 2025) ; in the tool-use setting, 15k documents from the Nemotron pretraining code corpus (NVIDIA, 2026). Section 6.2 shows how corpus grounding sustains environment diversity.

Environment memory. The corpus supplies breadth from outside the loop; the cross-episode memory keeps the `Environment Designer` from re-posing what the `Reasoning Agent` has already mastered (Schmidhuber, 2013). A buffer of previously generated environments, annotated with regret scores and skill tags, gives the `Environment Designer` high-regret seeds to vary, so each round starts from what the `Reasoning Agent` currently finds hard rather than from scratch. Retaining and reusing past experience this way is the mechanism agentic-memory systems rely on to keep improving without further gradient updates (Zhang et al., 2026d; Xiong et al., 2026). Environment pool management, lifecycle, and quality control details are in Appendix E.2.

5 Experimental Setup

`SPADE` Training Recipe. For games and tool use, we train three Qwen3 backbones: Qwen3-4B-Instruct-2507, Qwen3-8B, and Qwen3-30B-A3B-Instruct-2507, the last of which is our *primary* model. The 4B and 30B runs use instruct models; the 8B runs enable with thinking for both roles. We train with GRPO (Shao et al., 2024) for 400 rollouts of 24 environments each. The `Environment Designer` regenerates environments every k rollouts and delays its update by the same k (Section 4.1). The `Reasoning Agent` plays each environment 16 times with the hint and

Table 1: **SPADE improves held-out reasoning and code across model scales.** Metrics are Avg@32 for AIME 2025/2026, accuracy for GPQA-D, Pass@1 for LCB-v6, and win rate at HARD for Reasoning-Gym. Avg is the eight-benchmark mean; green subscripts show gains over base. Best and second-best results within each backbone are **bold** and underlined.

Model	AIME'25 (MAA, 2026)	AIME'26 (MAA, 2026)	GPQA-D (Rein et al., 2023)	LCB-v6 (Jain et al., 2024)	Reasoning-Gym (Stojanovski et al., 2025) RG-Math	RG-Algo.	RG-Cog.	RG-Logic	Avg	Δ vs Base
Qwen3 backbones: fixed-environment baselines and SPADE (game environment design)										
Qwen3-4B-Instruct-2507	47.4	58.3	55.9	35.1	31.6	11.8	16.4	54.7	38.9	–
Fixed-env GRPO	47.1	58.6	56.2	35.4	34.0	13.5	18.1	56.3	39.9	+1.0
Fixed-env RLVE	**49.6**	**62.1**	57.3	35.9	38.6	16.3	21.0	59.1	42.5	+3.6
+ SPADE (Games)	48.9$_{+1.5}$	60.2$_{+1.9}$	**58.1**$_{+2.2}$	**37.2**$_{+2.1}$	**44.6**$_{+13.0}$	**19.8**$_{+8.0}$	**23.1**$_{+6.7}$	**60.8**$_{+6.1}$	**44.1**	**+5.2**
Qwen3-8B	67.1	71.2	59.4	46.3	47.2	19.6	24.8	63.1	49.8	–
Fixed-env GRPO	67.4	71.0	59.9	46.8	49.8	21.9	26.5	64.6	51.0	+1.2
Fixed-env RLVE	**69.6**	**75.0**	61.6	47.4	52.7	25.0	31.0	67.9	53.8	+3.9
+ SPADE (Games)	68.8$_{+1.7}$	73.1$_{+1.9}$	**62.9**$_{+3.5}$	**49.4**$_{+3.1}$	**57.3**$_{+10.1}$	**29.2**$_{+9.6}$	**33.8**$_{+9.0}$	**69.7**$_{+6.6}$	**55.5**	**+5.7**
Qwen3-30B-A3B-Instruct-2507	61.5	73.5	70.4	43.2	45.0	18.0	23.0	67.0	50.2	–
Fixed-env GRPO	61.2	73.8	70.9	43.7	48.1	20.3	24.6	68.4	51.4	+1.2
Fixed-env RLVE	56.9	69.8	69.8	42.5	55.8	24.7	30.9	**73.7**	53.0	+2.8
+ SPADE (Games)	**62.8**$_{+1.3}$	**74.4**$_{+0.9}$	**75.8**$_{+5.4}$	**47.3**$_{+4.1}$	**63.3**$_{+18.3}$	**32.1**$_{+14.1}$	**37.7**$_{+14.7}$	72.8$_{+5.8}$	**58.3**	**+8.1**

$16\times k$ without, and the regret subtracts the two averages at the same regeneration step. We use the same stabilizers across settings and the **slime** RL framework (Zhu et al., 2025). Full hyperparameters and released configurations are described in Appendices E.2 and E.3.

5.1 Domain: Games

Environments. The `Environment Designer` writes each environment as a self-contained Python game with a verifiable reward, and the `Reasoning Agent` solves it. Each rollout draws 24 games from three of the six cognitive-skill categories (Mathematical Reasoning, Logical Deduction, Spatial Reasoning, Pattern Recognition, Optimization, and Causal Inference) in round-robin, grounded in the math and science corpus of Section 4.3. The `Environment Designer` reward blends floored hint-based regret (weight 0.4) with a flat-top difficulty anchor (weight 0.6), and every game is validated before entering the pool (Table 6; Appendix E.2).

Baselines and evaluation. We compare against two fixed-environment baselines, each retrained per backbone from its base checkpoint under the same 400-iteration budget: *Fixed-env RLVE* (Zeng et al., 2025) and *Fixed-env GRPO*, an offline pool the same backbone generates once and holds fixed. Evaluation spans two distances from the training distribution: *procedural reasoning* (the four Reasoning-Gym (Stojanovski et al., 2025) HARD categories) and *out-of-distribution reasoning and code* (AIME 2025/2026 (MAA, 2026), GPQA-Diamond (Rein et al., 2023), and LiveCodeBench-v6 (Jain et al., 2024)). Table 1 reports all eight benchmarks for every backbone; its caption gives the per-benchmark metrics, with full settings in Appendices E.2 and E.3.

5.2 Domain: Tool Use

Environments. The `Environment Designer` writes executable tool-use environments with simulated tools in OpenAI function-calling format, a backend state they modify, and three to five sequential natural-language user instructions with per-instruction state checks. Generation uses the 15k-document code corpus; the higher cost of tool environments motivates k=8. Beyond game validation, we add a deterministic reset gate and an LLM feasibility screen (Appendix E.2).

Baselines and evaluation. We compare `SPADE` with four synthetic-environment systems: AgentScaler (Fang et al., 2025a), Agent-World (Dong et al., 2026), Agent World Model (AWM) (Wang et al., 2026b), and EnvScaler (Song et al., 2026). Evaluation uses BFCL v4 multi-turn (Patil et al., 2025), τ^2-bench (Barres et al., 2025), and ACEBench-Agent (the Agent category of ACEBench-en) (Chen et al., 2025a).

5.3 Domain: Terminal

Environments. The `Environment Designer` generates terminal tasks with task instructions, sandbox setups, and executable verifiers, and the `Reasoning Agent` solves them via the Pi coding agent (Zechner, 2025). We adapt Tmax's task-generation recipe (Ivison et al., 2026), refreshing the training pool from the current checkpoint with k=8. Only tasks that build and pass validation enter training; hint-based regret rewards the task-design and verifier trajectories. Appendix E.2.1 gives the full protocol.

Table 2: **SPADE improves multi-step tool use across all backbones.** Per-domain results on BFCL v4 multi-turn, τ^2-bench, and ACEBench-Agent. Each Avg averages the shown subcolumns or uses the cited aggregate when subcolumns are unavailable; the final **Avg** averages all three benchmarks before rounding. '–' denotes unreported results; SPADE rows are **bold**.

Model	BFCL v4 (multi-turn)					τ^2-bench				ACEBench-Agent			Avg	Δ
	Base	Miss Func	Miss Param	Long Ctx	Avg	Retail	Airline	Telecom	Avg	Multi Step	Multi Turn	Avg		
Synthetic-environment agents (per-split numbers as reported by the cited papers; † suite labeled BFCL v3, see Appendix E.3.1)														
AgentScaler-30B-A3B (Fang et al., 2025a)	–	–	–	–	–	70.2	60.0	55.3	61.8	–	–	60.0	–	–
Agent-World-8B (Dong et al., 2026)	–	–	–	–	44.5	72.8	40.0	50.9	54.6	–	–	–	–	–
Agent-World-14B (Dong et al., 2026)	–	–	–	–	53.9	74.5	52.0	56.1	60.9	–	–	–	–	–
AWM-8B† (Wang et al., 2026b)	–	–	–	–	45.0	41.2	38.5	23.5	34.4	–	–	–	–	–
AWM-14B† (Wang et al., 2026b)	–	–	–	–	51.9	63.6	31.5	17.8	37.6	–	–	–	–	–
EnvScaler-4B† (Song et al., 2026)	51.0	34.0	28.0	39.0	38.0	–	–	–	–	80.0	61.1	70.6	–	–
EnvScaler-8B† (Song et al., 2026)	55.5	36.0	35.0	41.0	41.9	–	–	–	–	85.0	60.0	72.5	–	–
Ours: SPADE post-training on Qwen3 backbones (tool-use environment design)														
Qwen3-4B-Instruct-2507	34.0	16.0	12.5	25.5	22.0	43.0	32.0	18.0	31.0	55.0	41.7	48.4	33.8	
+ **SPADE ♠** (Tool Use)	**46.0**	**26.5**	**22.0**	**34.7**	**32.3**$_{+10.3}$	**47.2**	**35.6**	**21.5**	**34.8**$_{+3.8}$	**65.0**	**49.5**	**57.3**$_{+8.9}$	**41.4**	**+7.7**
Qwen3-8B	52.0	30.0	24.0	35.6	35.4	34.0	26.5	18.0	26.2	63.3	56.7	60.0	40.5	
+ **SPADE ♠** (Tool Use)	**58.0**	**36.0**	**30.0**	**43.2**	**41.8**$_{+6.4}$	**37.8**	**29.5**	**21.2**	**29.5**$_{+3.3}$	**73.0**	**65.0**	**69.0**$_{+9.0}$	**46.8**	**+6.2**
Qwen3-30B-A3B-Instruct-2507	66.0	44.0	38.0	48.0	49.0	62.0	50.0	35.0	49.0	70.0	54.0	62.0	53.3	
+ **SPADE ♠** (Tool Use)	**72.0**	**50.0**	**44.0**	**52.9**	**54.7**$_{+5.7}$	**65.5**	**53.5**	**38.8**	**52.6**$_{+3.6}$	**82.0**	**69.8**	**75.9**$_{+13.9}$	**61.1**	**+7.7**

Baselines and evaluation. All runs start from Nemotron-3.5-Lightning and we compare SPADE against Tmax, trained on the released Tmax-15k pool generated with Gemini-3-Pro; and SPADE without Environment Designer training, which still regenerates environments but excludes environment-design trajectories from optimization. Evaluation uses Terminal-Bench 2.1 (Merrill et al., 2026) and SWE-bench Verified (Jimenez et al., 2024).

6 Experimental Results

6.1 Quantitative Analysis

Synthetic game environments improve science, code, and procedural reasoning. At 30B-A3B, SPADE reaches a suite average of 58.3: +8.1 over base (Figure 5) and +5.3 over the strongest fixed-environment baseline (Fixed-env RLVE), a margin that grows with model size (Table 1). The Environment Designer trains on synthetic games, never on any held-out task, yet the gains transfer to science, code, and procedural reasoning and hold late in the 400-step run, while competition math is preserved. They concentrate on procedural reasoning, where every Reasoning-Gym category improves: the generated games exercise each skill through many problem structures rather than a fixed task set. The same recipe also improves Reasoning-Gym performance on a Nemotron model (Appendix F).

Synthetic tool-use environments improve multi-step interaction. The same recipe applied to tool-use environment design improves every backbone (Table 2), and at 30B-A3B the size of the gain tracks how closely a benchmark's task structure matches the generated environments: ACEBench-Agent gains most (+13.9), and its stateful, multi-step tasks mirror the generated pattern of a database, a tool schema, and a multi-call goal. BFCL v4 multi-turn follows (+5.7, and +10.3 at 4B), then τ^2-bench (+3.6). SPADE at 30B-A3B leads the dedicated data-synthesis systems on both BFCL v4 multi-turn and ACEBench-Agent, so structural training signal transfers where domain-specific data collection does not reach.

Synthetic terminal environments improve coding performance. On Nemotron-3.5-Lightning, SPADE reaches 31.4 on Terminal-Bench 2.1 and 57.2 on SWE-bench Verified, gaining +7.2 and +8.0 points over base (Table 3). Its mean scores exceed those of the variant without Environment Designer training by +6.1 and +3.8 points, respectively, showing the importance of evolving Environment Designer.

Table 3: **Terminal-agent results.** Terminal-Bench 2.1 is reported as Avg@5; SWE-bench Verified reports the percentage of resolved tasks.

Model	TB2.1	SWE-V
Nemotron-3.5-Lightning	24.2 ± 4.3	49.2 ± 1.8
+ Tmax (Ivison et al., 2026)	<u>28.8</u> ± 3.2	<u>55.8</u> ± 2.3
+ **SPADE ♠** (w/o ED training)	25.3 ± 2.1	53.4 ± 2.3
+ **SPADE ♠**	**31.4**$_{+7.2}$ ±2.9	**57.2**$_{+8.0}$ ±2.5

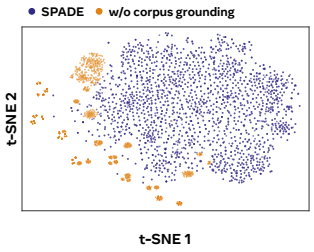


Figure 4: **Grounding preserves diversity.** Vendi/n 0.68 with grounding versus 0.04 without.

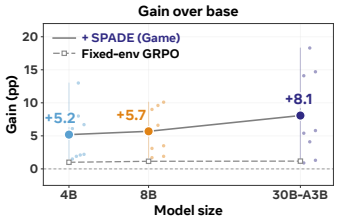


Figure 5: **`SPADE` scales with model size.** Gain over each backbone's base.

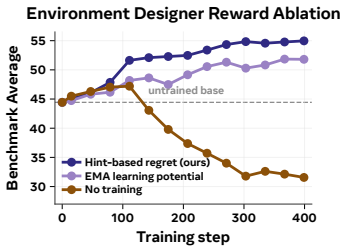


Figure 6: **Environment reward ablation.** Mean of the six non-AIME benchmarks.

6.2 Qualitative Analysis

Corpus grounding preserves environment diversity. With corpus grounding, generated environments span the embedding space; without it, they collapse into tight clusters (Figure 4). The normalized Vendi scores confirm this difference: 0.68 with grounding versus 0.04 without, where a higher score indicates a more diverse environment set. Further analyses of environment difficulty and `Environment Designer` and `Reasoning Agent` outputs are in Appendices F.5 and G.1.2.

6.3 Ablations

We ablate the `Environment Designer`'s inputs (corpus and memory), whether it is trained, and how it is rewarded. All experiments use games on Qwen3-30B-A3B-Instruct-2507 and are evaluated on the same benchmarks as Table 1. Each variant reports its best checkpoint on the suite average; full curves and per-benchmark results are in Appendix F.

`Environment Designer` Reward. We compare our hint-based regret with EMA-based learning potential (Kanitscheider et al., 2021; Zhang et al., 2023). This alternative measures the absolute deviation of the `Reasoning Agent`'s success rate on each environment from its skill's exponential moving average, aiming to keep generation near the learner's frontier. Hint-based regret achieves a best-checkpoint eight-benchmark average of 58.3, versus 55.9 for EMA-based learning potential. Appendix E.2 provides the full reward definition; Figure 6 shows six-benchmark training curves, and Appendix F gives per-benchmark results.

`Environment Designer` Ablation. Table 4 compares `SPADE` with variants that remove environment memory, corpus grounding, or both `Environment Designer` training and memory, as well as a variant that uses a static environment pool generated by GPT-5.5. Removing either memory or corpus grounding substantially reduces the gain, while removing both `Environment Designer` training and memory ends below the base model. The GPT-5.5 pool improves over the base model but trails every variant that trains the designer. Additional curves and per-benchmark results are in Appendix F.

Table 4: **Environment Designer ablations.** Best-checkpoint average across eight benchmarks; per-benchmark results are in Table 10.

Setting	ED model	ED trained	Corpus	Env. memory	Benchmarks Avg
Base model	–	–	–	–	50.2
SPADE ♠	Self	✓	✓	✓	**58.3**
w/o memory	Self	✓	✓	✗	53.2
w/o corpus	Self	✓	✗	✓	53.5
w/o ED training	Self	✗	✓	✗	40.5
Fixed ED (GPT-5.5)	GPT-5.5	✗	✓	✗	53.0

7 Conclusion

`SPADE` makes environment design a learnable part of LLM post-training: one model writes executable environments and learns to solve them, while hint-based regret directs generation toward the learner's current frontier. Our results present two takeaways. First, training the `Environment Designer` to adapt environments outperforms training on a fixed pool. Second, the same code-as-environment interface supports both reasoning games and multi-step tool use, with capabilities learned in these environments transferring beyond the training tasks. Together, these findings show that a model can design its own training environments and improve from them, offering a practical step from fixed curricula toward open-ended self-improvement.

AI USE STATEMENT

Language models are part of the method: they generate executable environments, reward and verification code, and privileged hints, and act within the generated environments. Model-based screening and the programmatic checks are described in Section 5 and Appendix E.2. We also use LLM rubrics to assess environment well-posedness and verifiability, and LLM-generated domain labels in the diversity analysis (Appendices F.5 and F.6); these assessments are model judgments, not independent human annotations. AI assistants were used for code and manuscript editing, literature lookup, and formatting of result tables and figures. The authors remain responsible for the manuscript and its claims.

ETHICS STATEMENT

SPADE studies self-improvement in synthetic games and simulated tool-use environments. Improved reasoning and tool-use capabilities can have beneficial applications, but can also support harmful automation. Benchmark gains do not establish safe behavior in deployment, and generated environments or verifiers may contain errors, exploitable rewards, or unsafe code. Executability and solvability checks do not constitute a security audit; applications should isolate generated code, restrict permissions, and avoid exposing credentials or sensitive data. Corpus grounding and pretrained models can propagate biases, copyrighted material, or personal information into synthetic environments, so generated data require appropriate review before release or reuse. The LLM-based quality assessments may also reproduce model biases. Finally, repeated generation, rollout, and training incur computational and environmental costs; the experimental scope and training configurations are reported to help readers assess these tradeoffs.

REPRODUCIBILITY STATEMENT

Section 4 specifies the learning procedure, and Section 5 describes the models, training settings, baselines, and evaluation protocol. Appendix D provides the theoretical assumptions and proofs. The environment-generation prompts appear in Appendix H and a minimal environment in Appendix G.3.1, while Appendices E.2 and E.3 describe the generation and validation lifecycle, implementation details, and reproducibility settings. Appendix F provides additional results and analysis details. These descriptions distinguish programmatic measurements from the model-based assessments disclosed above.

REFERENCES

Marwa Abdulhai, Isadora White, Yanming Wan, Ibrahim Qureshi, Joel Leibo, Max Kleiman-Weiner, and Natasha Jaques. How LLMs distort our written language, 2026. URL https://arxiv.org/abs/2603.18161.

Emre Can Acikgoz, Cheng Qian, Jonas Hübotter, Heng Ji, Dilek Hakkani-Tür, and Gokhan Tur. Tool-r0: Self-evolving llm agents for tool-learning from zero data. *arXiv preprint arXiv:2602.21320*, 2026.

Luke Bailey, Kaiyue Wen, Kefan Dong, Tatsunori Hashimoto, and Tengyu Ma. Scaling self-play with self-guidance. *arXiv preprint arXiv:2604.20209*, 2026.

Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. In *International Conference on Learning Representations*, 2020.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pp. 41–48, 2009.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Adrian Bolton, Alexander Lerchner, Alexandra Cordell, Alexandre Moufarek, Andrew Bolt, Andrew Lampinen, Anna Mitenkova, Arne Olav Hallingstad, Bojan Vujatovic, Bonnie Li, et al. Sima 2: A generalist embodied agent for virtual worlds. *arXiv preprint arXiv:2512.04797*, 2025.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Roger Creus Castanyer, Geoffrey Bradway, Lorenz Wolf, Maxwill Lin, Augustine N Mavor-Parker, and Matthew James Sargent. Populora: Co-evolving llm populations for reasoning self-play. *arXiv preprint arXiv:2605.16727*, 2026.

Zhepeng Cen, Haolin Chen, Shiyu Wang, Zuxin Liu, Zhiwei Liu, Ding Zhao, Caiming Xiong, Huan Wang, and Weiran Yao. Webscale-RL: Automated data pipeline for scaling RL data to pretraining levels. In *International Conference on Learning Representations*, volume 2026, pp. 87623–87642, 2026.

Justin Yang Chae, Md Tanvirul Alam, and Nidhi Rastogi. Towards understanding self-play for llm reasoning. *arXiv preprint arXiv:2510.27072*, 2025.

Chao Chen, Chengzu Li, Zhiwei Li, Yinhong Liu, and Zhijiang Guo. From trainee to trainer: Llm-designed training environment for rl with multi-agent reasoning. *arXiv preprint arXiv:2606.17682*, 2026a.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, et al. Acebench: Who wins the match point in tool usage? *arXiv preprint arXiv:2501.12851*, 2025a.

Jiaqi Chen, Bang Zhang, Ruotian Ma, Peisong Wang, Xiaodan Liang, Zhaopeng Tu, Xiaolong Li, and Kwan-Yee K Wong. SPC: Evolving self-play critic via adversarial games for LLM reasoning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 139228–139254, 2025b.

Lili Chen, Mihir Prabhudesai, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak. Self-questioning language models. *arXiv preprint arXiv:2508.03682*, 2025c.

Xiaoyin Chen, Jiarui Lu, Minsu Kim, Dinghuai Zhang, Jian Tang, Alexandre Piché, Nicolas Gontier, Yoshua Bengio, and Ehsan Kamalloo. Self-evolving curriculum for llm reasoning. *arXiv preprint arXiv:2505.14970*, 2025d.

Zhaorun Chen, Zhuokai Zhao, Kai Zhang, Bo Liu, Qi Qi, Yifan Wu, Tarun Kalluri, Xuefei Cao, Yuanhao Xiong, Haibo Tong, Huaxiu Yao, Hengduo Li, Jiacheng Zhu, Xian Li, Dawn Song, Bo Li, Jason E Weston, and Dat Huynh. Scaling agent learning via experience synthesis. In *International Conference on Learning Representations*, volume 2026, pp. 121394–121420, 2026b.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.

Daixuan Cheng, Shaohan Huang, Yuxian Gu, Huatong Song, Guoxin Chen, Li Dong, Wayne Xin Zhao, Ji-Rong Wen, and Furu Wei. Computer environments elicit general agentic intelligence in LLMs. *arXiv preprint arXiv:2601.16206*, 2026.

Pengyu Cheng, Yong Dai, Tianhao Hu, Han Xu, Zhisong Zhang, Lei Han, Nan Du, and Xiaolong Li. Self-playing adversarial language game enhances llm reasoning. *Advances in Neural Information Processing Systems*, 37:126515–126543, 2024.

Caroline Choi, Zeyneb Kaya, Shirley Wu, Tengyu Ma, Tatsunori Hashimoto, and Ludwig Schmidt. Anchored self-play for code repair. *arXiv preprint arXiv:2607.03523*, 2026.

Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

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

Leander Diaz-Bone, Marco Bagatella, Jonas Hübotter, and Andreas Krause. DISCOVER: Automated curricula for sparse-reward reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 21863–21896, 2025.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. Self-play with execution feedback: Improving instruction-following capabilities of large language models. *arXiv preprint arXiv:2406.13542*, 2024.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. Omni-epic: Open-endedness via models of human notions of interestingness with environments programmed in code. *arXiv preprint arXiv:2405.15568*, 2024.

Run-Ze Fan, Zengzhi Wang, and Pengfei Liu. Megascience: Pushing the frontiers of post-training datasets for science reasoning. *arXiv preprint arXiv:2507.16812*, 2025.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, et al. Toward scalable terminal task synthesis via skill graphs. *arXiv preprint arXiv:2604.25727*, 2026.

Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, et al. Towards general agentic intelligence via environment scaling. *arXiv preprint arXiv:2509.13311*, 2025a.

Wenkai Fang, Shunyu Liu, Yang Zhou, Kongcheng Zhang, Tongya Zheng, Kaixuan Chen, Mingli Song, and Dacheng Tao. SeRL: Self-play reinforcement learning for large language models with limited data. In *Advances in Neural Information Processing Systems*, volume 38, pp. 103706–103738, 2025b.

Dan Friedman and Adji Bousso Dieng. The Vendi score: A diversity evaluation metric for machine learning, 2023. URL https://arxiv.org/abs/2210.02410.

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

Judah Goldfeder, Philippe Wyder, Yann LeCun, and Ravid Shwartz Ziv. Ai must embrace specialization via superhuman adaptable intelligence. *arXiv preprint arXiv:2602.23643*, 2026.

Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. Textarena. *arXiv preprint arXiv:2504.11442*, 2025.

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. *arXiv preprint arXiv:2512.19682*, 2025.

Ahmet H Güzel, Matthew Thomas Jackson, Jarek Luca Liesen, Tim Rocktäschel, Jakob Nicolaus Foerster, Ilija Bogunovic, and Jack Parker-Holder. Imagined autocurricula. In *Advances in Neural Information Processing Systems*, volume 38, pp. 114787–114808, 2025.

Chengsong Huang, Haolin Liu, Tong Zheng, Runpeng Dai, Langlin Huang, Jinyuan Li, Zongxia Li, Zhepei Wei, Yu Meng, and Jiaxin Huang. G-zero: Self-play for open-ended generation from zero data. *arXiv preprint arXiv:2605.09959*, 2026a.

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. In *International Conference on Learning Representations*, volume 2026, pp. 130770–130790, 2026b.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktaschel. Open-endedness is essential for artificial superhuman intelligence. *arXiv preprint arXiv:2406.04268*, 2024.

Geoffrey Irving, Paul Christiano, and Dario Amodei. Ai safety via debate. *arXiv preprint arXiv:1805.00899*, 2018.

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Swadesh Jana, Cansu Sancaktar, Tomáš Daniš, Georg Martius, Antonio Orvieto, and Pavel Kolev. Gasp: Guided asymmetric self-play for coding llms. *arXiv preprint arXiv:2603.15957*, 2026.

Liwei Jiang, Yuanjun Chai, Margaret Li, Mickel Liu, Raymond Fok, Nouha Dziri, Yulia Tsvetkov, Maarten Sap, and Yejin Choi. Artificial hivemind: The open-ended homogeneity of language models (and beyond). In *Advances in Neural Information Processing Systems*, volume 38, 2025a.

Minqi Jiang, Michael Dennis, Jack Parker-Holder, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Replay-guided adversarial environment design. *Advances in Neural Information Processing Systems*, 34:1884–1897, 2021a.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, pp. 4940–4950. PMLR, 2021b.

Minqi Jiang, Andrei Lupu, and Yoram Bachrach. Bootstrapping task spaces for self-improvement. *arXiv preprint arXiv:2509.04575*, 2025b.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Ingmar Kanitscheider, Joost Huizinga, David Farhi, William Hebgen Guss, Brandon Houghton, Raul Sampedro, Peter Zhokhov, Bowen Baker, Adrien Ecoffet, Jie Tang, Oleg Klimov, and Jeff Clune. Multi-task curriculum learning in a complex, visual, hard-exploration domain: Minecraft, 2021. URL https://arxiv.org/abs/2106.14876.

Jan Hendrik Kirchner, Yining Chen, Harri Edwards, Jan Leike, Nat McAleese, and Yuri Burda. Prover-verifier games improve legibility of llm outputs. *arXiv preprint arXiv:2407.13692*, 2024.

Jakub Grudzien Kuba, Mengting Gu, Qi Ma, Yuandong Tian, Vijai Mohan, and Jason Chen. Language self-play for data-free training. *arXiv preprint arXiv:2509.07414*, 2025.

Ilia Kulikov, Chenxi Whitehouse, Tianhao Wu, Yixin Nie, Swarnadeep Saha, Eryk Helenowski, Weizhe Yuan, Olga Golovneva, Jack Lanchantin, Yoram Bachrach, et al. Autodata: An agentic data scientist to create high quality synthetic data. *arXiv preprint arXiv:2606.25996*, 2026.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. Datacomp-lm: In search of the next generation of training sets for language models. *arXiv preprint arXiv:2406.11794*, 2024.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. Simulating environments with reasoning models for agent training. *arXiv preprint arXiv:2511.01824*, 2025.

William Liang, Sam Wang, Hung-Ju Wang, Osbert Bastani, Dinesh Jayaraman, and Yecheng Jason Ma. Eurekaverse: Environment curriculum generation via large language models. *arXiv preprint arXiv:2411.01775*, 2024.

Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pp. 157–163. Elsevier, 1994.

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

Bo Liu, Simon Yu, Zichen Liu, Leon Guertler, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, Wee Sun Lee, and Natasha Jaques. SPIRAL: Self-play on zero-sum games incentivizes reasoning via multi-agent multi-turn reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 35407–35434, 2026a.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Peter Belcak, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Gdpo: Group reward-decoupled normalization policy optimization for multi-reward rl optimization, 2026b. URL https://arxiv.org/abs/2601.05242.

Wei Liu, Siya Qi, Yali Du, and Yulan He. Self-play only evolves when self-synthetic pipeline ensures learnable information gain. *arXiv preprint arXiv:2603.02218*, 2026c.

Zichen Liu, Anya Sims, Keyu Duan, Changyu Chen, Simon Yu, Xiangxin Zhou, Haotian Xu, Shaopan Xiong, Bo Liu, Chenmien Tan, et al. Gem: A gym for agentic llms. *arXiv preprint arXiv:2510.01051*, 2025b.

Siyuan Lu, Zechuan Wang, Hongxuan Zhang, Qintong Wu, Leilei Gan, Chenyi Zhuang, Jinjie Gu, and Tao Lin. Don't just fine-tune the agent, tune the environment. In *International Conference on Learning Representations*, volume 2026, pp. 6237–6283, 2026a.

Ximing Lu, David Acuna, Jaehun Jung, Jian Hu, Di Zhang, Shizhe Diao, Yunheng Zou, Shaokun Zhang, Brandon Cui, Mingjie Liu, et al. Golden goose: A simple trick to synthesize unlimited rlvr tasks from unverifiable internet text. *arXiv preprint arXiv:2601.22975*, 2026b.

MAA. American Invitational Mathematics Examination (AIME), 2026. URL https://maa.org/maa-invitational-competitions/. AIME 2025 and AIME 2026.

Ishita Mediratta, Minqi Jiang, Jack Parker-Holder, Michael Dennis, Eugene Vinitsky, and Tim Rocktäschel. Stabilizing unsupervised environment design with a learned adversary. In *Conference on Lifelong Learning Agents*, pp. 270–291. PMLR, 2023.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Meta Fundamental AI Research Diplomacy Team (FAIR)†, Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, Athul Paul Jacob, Mojtaba Komeili, Karthik Konath, Minae Kwon, Adam Lerer, Mike Lewis, Alexander H. Miller, Sasha Mitts, Adithya Renduchintala, Stephen Roller, Dirk Rowe, Weiyan Shi, Joe Spisak, Alexander Wei, David Wu, Hugh Zhang, and Markus Zijlstra. Human-level play in the game of *Diplomacy* by combining language models with strategic reasoning. *Science*, 378(6624): 1067–1074, 2022. doi: 10.1126/science.ade9097. URL https://www.science.org/doi/abs/10.1126/science.ade9097.

Nathan Monette, Alistair Letcher, Michael Beukman, Matthew T Jackson, Alexander Rutherford, Alexander D Goldie, and Jakob N Foerster. An optimisation framework for unsupervised environment design. *arXiv preprint arXiv:2505.20659*, 2025.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

NVIDIA. Nemotron 3 ultra: Open, efficient mixture-of-experts hybrid mamba-transformer model for agentic reasoning, 2026. URL https://research.nvidia.com/labs/nemotron/files/NVIDIA-Nemotron-3-Ultra-Technical-Report.pdf. White Paper.

OpenAI. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. In *International Conference on Machine Learning*, pp. 17473–17498. PMLR, 2022.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*, 2025.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling llm terminal capabilities. *arXiv preprint arXiv:2602.21193*, 2026.

PYMNTS. Prime Intellect Raises $130 Million to Help Companies Train AI Agents. https://www.pymnts.com/news/investment-tracker/2026/prime-intellect-raises-130-million-to-help-companies-train-ai-agents/, 2026. PYMNTS.

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

Ram Ramrakhya, Andrew Szot, Omar Attia, Bogdan Mazoure, Anh Nguyen, Yuhao Yang, Zhe Gan, Harsh Agrawal, and Alexander Toshev. Scaling synthetic task generation for agents via exploration. In *International Conference on Learning Representations*, volume 2026, pp. 123630–123660, 2026.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks, 2019. URL https://arxiv.org/abs/1908.10084.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Alex Rutherford, Michael Beukman, Timon Willi, Bruno Lacerda, Nick Hawes, and Jakob Foerster. No regrets: Investigating and improving regret approximations for curriculum discovery. *Advances in Neural Information Processing Systems*, 37:16071–16101, 2024.

Bidipta Sarkar, Warren Xia, C Karen Liu, and Dorsa Sadigh. Training language models for social deduction with multi-agent reinforcement learning. *arXiv preprint arXiv:2502.06060*, 2025.

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

Sheikh Shafayat, Fahim Tajwar, Ruslan Salakhutdinov, Jeff Schneider, and Andrea Zanette. Can large reasoning models self-train? *arXiv preprint arXiv:2505.21444*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Yucheng Shi, Zhenwen Liang, Kishan Panaganti, Dian Yu, Wenhao Yu, and Haitao Mi. Learning to build the environment: Self-evolving reasoning rl via verifiable environment synthesis. *arXiv preprint arXiv:2605.14392*, 2026.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://arxiv.org/abs/2303.11366.

David Silver and Richard S. Sutton. Welcome to the era of experience, 2025.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, 2018.

Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. Envscaler: Scaling tool-interactive environments for llm agent via programmatic synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, 2026.

Yuda Song, Hanlin Zhang, Carson Eisenach, Sham Kakade, Dean Foster, and Udaya Ghai. Mind the gap: Examining the self-improvement capabilities of large language models. *arXiv preprint arXiv:2412.02674*, 2024.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning Gym: Reasoning environments for reinforcement learning with verifiable rewards. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

Shobhita Sundaram, John Quan, Ariel Kwiatkowski, Kartik Ahuja, Yann Ollivier, and Julia Kempe. Teaching models to teach themselves: Reasoning at the edge of learnability. *arXiv preprint arXiv:2601.18778*, 2026.

Fahim Tajwar, Yiding Jiang, Abitha Thankaraj, Sumaita Sadia Rahman, J Zico Kolter, Jeff Schneider, and Ruslan Salakhutdinov. Training a generally curious agent. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 58227–58281, 2025.

Miles Team. Miles: Enterprise-grade reinforcement learning for large-scale model post-training. https://github.com/radixark/miles, 2026.

Jayden Teoh, Wenjun Li, and Pradeep Varakantham. Improving environment novelty quantification for effective unsupervised environment design. *Advances in Neural Information Processing Systems*, 37:135299–135333, 2024.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38 (3):58–68, 1995.

Mark Towers, Ariel Kwiatkowski, John Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Kallinteris Andreas, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, et al. Gymnasium: A standard interface for reinforcement learning environments. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, et al. Scaleenv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training. *arXiv preprint arXiv:2602.06820*, 2026.

Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. Will we run out of data? Limits of LLM scaling based on human-generated data. *arXiv preprint arXiv:2211.04325*, 2024. URL https://arxiv.org/abs/2211.04325.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.

Ziyu Wan, Yunxiang Li, Xiaoyu Wen, Yan Song, Hanjing Wang, Linyi Yang, Mark Schmidt, Jun Wang, Weinan Zhang, Shuyue Hu, et al. ReMA: Learning to meta-think for LLMs with multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 126621–126667, 2025.

Qinsi Wang, Bo Liu, Tianyi Zhou, Jing Shi, Yueqian Lin, Yiran Chen, Hai Helen Li, Kun Wan, and Wentian Zhao. Vision-zero: Scalable vlm self-improvement via strategic gamified self-play. *arXiv preprint arXiv:2509.25541*, 2025a.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. Rlanything: Forge environment, policy, and reward model in completely dynamic rl system. *arXiv preprint arXiv:2602.02488*, 2026a.

Yuxing Wang, Zhiyu Chen, Tiantian Zhang, Qiyue Yin, Yongzhe Chang, Zhiheng Li, Liang Wang, and Xueqian Wang. Embodied co-design for rapidly evolving agents: Taxonomy, frontiers, and challenges. *arXiv preprint arXiv:2512.04770*, 2025b.

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. *arXiv preprint arXiv:2602.10090*, 2026b.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025c.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, volume 38, pp. 78500–78525, 2025a.

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

Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. *arXiv preprint arXiv:2602.07755*, 2026.

Caijun Xu, Changyi Xiao, Zhongyuan Peng, Xinrun Wang, and Yixin Cao. Scaler: Synthetic scalable adaptive learning environment for reasoning. *arXiv preprint arXiv:2601.04809*, 2026a.

Fengli Xu, Qianyue Hao, Chenyang Shao, Zefang Zong, Yu Li, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, et al. Toward large reasoning models: A survey of reinforced reasoning with large language models. *Patterns*, 6(10), 2025.

Minrui Xu, Zilin Wang, Mengyi Deng, Zhiwei Li, Zhicheng Yang, Xiao Zhu, Yinhong Liu, Boyu Zhu, Baiyu Huang, Chao Chen, et al. Envfactory: Scaling tool-use agents via executable environments synthesis and robust rl. *arXiv preprint arXiv:2605.18703*, 2026b.

Taofeng Xue, Chong Peng, Mianqiu Huang, Linsen Guo, Tiancheng Han, Haozhe Wang, Jianing Wang, Xiaocheng Zhang, Xin Yang, Dengchang Zhao, et al. Evocua: Evolving computer use agents via learning from scalable synthetic experience. *arXiv preprint arXiv:2601.15876*, 2026a.

Tianci Xue, Zeyi Liao, Tianneng Shi, Zilu Wang, Kai Zhang, Dawn Song, Yu Su, and Huan Sun. Autonomous continual learning for environment adaptation of computer-use agents. *arXiv preprint arXiv:2602.10356*, 2026b.

Chengyi Yang, Zhishang Xiang, Yunbo Tang, Zongpei Teng, Chengsong Huang, Fei Long, Yuhan Liu, and Jinsong Su. Ttcs: Test-time curriculum synthesis for self-evolving. *arXiv preprint arXiv:2601.22628*, 2026a.

Ziyi Yang, Weizhou Shen, Chenliang Li, Ruijun Chen, Fanqi Wan, Ming Yan, Xiaojun Quan, and Fei Huang. SPELL: Self-play reinforcement learning for evolving long-context language models. In *International Conference on Learning Representations*, volume 2026, pp. 59910–59942, 2026b.

Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. Your efficient rl framework secretly brings you off-policy rl training, August 2025. URL https://fengyao.notion.site/off-policy-rl.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2210.03629.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems*, volume 38, pp. 113222–113244, 2025.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason E Weston. Self-rewarding language models. In *Forty-first International Conference on Machine Learning*, 2024.

Mario Zechner. What I learned building an opinionated and minimal coding agent. Blog post, November 2025. URL https://mariozechner.at/posts/2025-11-30-pi-coding-agent/.

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

Zhiyuan Zeng, Hamish Ivison, Yiping Wang, Lifan Yuan, Shuyue Stella Li, Zhuorui Ye, Siting Li, Jacqueline He, Runlong Zhou, Tong Chen, et al. Rlve: Scaling up reinforcement learning for language models with adaptive verifiable environments. *arXiv preprint arXiv:2511.07317*, 2025.

Hanchen Zhang, Xiao Liu, Bowen Lv, Xueqiao Sun, Bohao Jing, Iat Long Iong, Zhenyu Hou, Zehan Qi, Hanyu Lai, Yifan Xu, et al. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework. *arXiv preprint arXiv:2510.04206*, 2025a.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. Omni: Open-endedness via models of human notions of interestingness. *arXiv preprint arXiv:2306.01711*, 2023.

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026a.

Jiayi Zhang, Yiran Peng, Fanqi Kong, Cheng Yang, Yifan Wu, Zhaoyang Yu, Jinyu Xiang, Jianhao Ruan, Jinlin Wang, Maojia Song, et al. Autoenv: Automated environments for measuring cross-environment agent learning. *arXiv preprint arXiv:2511.19304*, 2025b.

Jiayi Zhang, Fanqi Kong, Guibin Zhang, Maojia Song, Zhaoyang Yu, Jianhao Ruan, Jinyu Xiang, Bang Liu, Chenglin Wu, and Yuyu Luo. Scalable environments drive generalizable agents. *arXiv preprint arXiv:2605.18181*, 2026b.

Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R. Tomz, Christopher D. Manning, and Weiyan Shi. Verbalized sampling: How to mitigate mode collapse and unlock llm diversity, 2026c. URL https://arxiv.org/abs/2510.01171.

Kai Zhang, Xiangchao Chen, Bo Liu, Tianci Xue, Zeyi Liao, Zhihan Liu, Xiyao Wang, Yuting Ning, Zhaorun Chen, Xiaohan Fu, et al. Agent learning via early experience. *arXiv preprint arXiv:2510.08558*, 2025c.

Shengtao Zhang, Jiaqian Wang, Ruiwen Zhou, Junwei Liao, Yuchen Feng, Zhuo Li, Yujie Zheng, Weinan Zhang, Ying Wen, Zhiyu Li, et al. Memrl: Self-evolving agents via runtime reinforcement learning on episodic memory. *arXiv preprint arXiv:2601.03192*, 2026d.

Zhengxin Zhang, Chengyu Huang, Aochong Oliver Li, and Claire Cardie. Better llm reasoning via dual-play. *arXiv preprint arXiv:2511.11881*, 2025d.

Ziyun Zhang, Zezhou Wang, Xiaoyi Zhang, Zongyu Guo, Jiahao Li, Bin Li, and Yan Lu. Infiniteweb: Scalable web environment synthesis for gui agent training. *arXiv preprint arXiv:2601.04126*, 2026e.

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

Yifei Zhou, Sergey Levine, Jason Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents. In *Advances in Neural Information Processing Systems*, volume 38, pp. 113959–113991, 2025.

Yuhang Zhou, Lizhu Zhang, Yifan Wu, Jiayi Liu, Xiangjun Fan, Zhuokai Zhao, and Hong Yan. Synthetic sandbox for training machine learning engineering agents. *arXiv preprint arXiv:2604.04872*, 2026.

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

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

A Limitations and Future Directions

A.1 Limitations

Complexity is bounded by scale and the invisible leash (Chae et al., 2025). The `Environment Designer` cannot produce environments more complex than its base model can express in context, so reachable environment complexity grows with model scale and generation budget. **The optimizer is human-designed.** Both roles are updated by a fixed, human-authored RL algorithm (GRPO); `SPADE` does not modify its own learning rule. **There is no formal optimality guarantee, and evaluation uses fixed tasks.** Hint-based regret is motivated by PAIRED (Dennis et al., 2020) but not proven to yield an optimal curriculum, and current benchmarks measure fixed-task performance rather than open-ended reasoning growth.

A.2 Future Directions

We improve the `Environment Designer` with gradient updates, but co-adaptation between the `Environment Designer` and `Reasoning Agent` might also come from an `Environment Designer` that improves without weight updates, accumulating and refining design strategies from past attempts in context; whether learned weights or in-context evolution makes the better designer, and at what scale, is an open question. `SPADE` also automates one stage of post-training; combining adaptive environment design with systems that automate the remaining stages, from data curation through the learning rule itself, could extend co-adaptive self-play across the whole training pipeline.

B Extended Related Work

B.1 Self-Play for LLMs

Self-play has been a cornerstone of AI since Tesauro (1995) trained TD-Gammon through self-play in backgammon. AlphaGo (Silver et al., 2016) combined Monte Carlo tree search with deep neural networks and self-play to defeat a human Go champion, and AlphaZero (Silver et al., 2017; 2018) extended the approach to chess, shogi, and Go from tabula rasa with no human data. OpenAI Five (Berner et al., 2019) and AlphaStar (Vinyals et al., 2019) scaled multi-agent self-play to Dota 2 and StarCraft II, while Cicero (Meta Fundamental AI Research Diplomacy Team (FAIR)† et al., 2022) combined language models with strategic reasoning in Diplomacy. Littman (1994) formalized the Markov-game framework underlying these results, and Irving et al. (2018) proposed debate as an AI-safety mechanism grounded in self-play. The asymmetric self-play paradigm of Sukhbaatar et al. (2017), in which one agent proposes challenges while another solves them, provides the conceptual template for teacher-student environment design. PowerPlay (Schmidhuber, 2013) proposed a single self-modifying system that continually invents tasks at its own capability frontier, prefiguring the dual-role single-LLM pattern that `SPADE` adopts.

Translating self-play to LLMs introduces new challenges because rewards are sparse, outputs are discrete sequences, and the "environment" is open-ended language. SPIN (Chen et al., 2024) applies self-play fine-tuning by training the LLM to distinguish its own outputs from human demonstrations, converting weak models to strong ones. Self-Rewarding Language Models (Yuan et al., 2024) let the LLM judge its own outputs to generate preference data, iteratively improving both generation and evaluation. SPAG (Cheng et al., 2024) uses adversarial language games (Adversarial Taboo) to incentivize strategic reasoning through a zero-sum objective. ReST$^{\mathrm{EM}}$ (Singh et al., 2023) alternates between sampling model solutions filtered by a binary correctness reward and supervised fine-tuning on the filtered correct rollouts. Prover-Verifier Games (Kirchner et al., 2024) train a prover to produce legible solutions that a weaker verifier can check, improving output interpretability. ReMA (Wan et al., 2025) decomposes reasoning via multi-agent RL into a high-level meta-thinking agent (strategy and decomposition) and a low-level reasoning agent (step-by-step execution), trained jointly.

The most relevant line of recent work uses LLMs as their own source of training data in self-play, with varying degrees of reliance on external data. At one extreme, fully data-free methods bootstrap from minimal seeds: Absolute Zero Reasoner (AZR) (Zhao et al., 2025) has a single model (shared parameters) propose and solve code-reasoning triplets (deduction, abduction, induction) with proposer reward $1-\bar{r}_{\text{solve}}$ computed from solver Monte-Carlo success rates verified by a code executor,

bootstrapping from a single identity function. R-Zero (Huang et al., 2026b) instead instantiates a Challenger and a Solver as two independently optimized copies of the same base LLM, training the Challenger via GRPO with a self-consistency uncertainty reward; the paper reports that this two-model design outperforms a single-model variant in ablations but plateaus after a few iterations. PopuLoRA (Castanyer et al., 2026) extends this asymmetric paradigm to co-evolving populations of LoRA-adapter teachers and students with cross-evaluation between sub-populations. In a related but mechanistically distinct line, G-Zero (Huang et al., 2026a) performs DPO-based self-distillation over hint-induced preference pairs: a Proposer trained via GRPO emits a hint h for query q, the Generator π_G produces both an unassisted response $a_{\text{hard}} \sim \pi_G(\cdot \mid q)$ and a hint-conditional response $a_{\text{assisted}} \sim \pi_G(\cdot \mid q, h)$, and DPO is applied with a_{assisted} as chosen and a_{hard} as rejected; the Proposer's reward is Hint-δ, the per-token mean log-probability shift induced on a_{hard} when the hint is prepended to the Generator's context. All supervision is derived from a single model's own outputs under hint vs no-hint contexts, placing the method closer to the self-rewarding and self-distillation line (Chen et al., 2024; Yuan et al., 2024; Singh et al., 2023) than to verifier-grounded agent-environment co-evolution. Tool-R0 (Acikgoz et al., 2026) adapts the same Challenger-Solver paradigm to tool use, training the generator with a solve-rate-band reward plus a semantic alignment score and the solver with soft tool-call matching, similarly finding that separate parameters outperform a shared-weight setup. Self-Questioning Language Models (Chen et al., 2025c) have a single model propose and solve problems from only a topic prompt, with the proposer rewarded for majority-vote calibration (problems neither always solved nor always missed). Language Self-Play (Kuba et al., 2025) proposes a data-free framework where question generation and answer improvement reinforce each other. PasoDoble (Zhang et al., 2025d) pairs an adversarial proposer and solver grounded in a pretraining knowledge base, sustaining improvement beyond R-Zero's plateau. At the other extreme, methods that retain limited seed data or curated evaluation sets include SeRL (Fang et al., 2025b), which uses a few-shot-prompted question generator with a difficulty-band filter and trains only the solver via Reinforce++ on majority-vote rewards from 500 seed instructions, and Sundaram et al. (2026), who ground the teacher's reward in measured student improvement on curated target problems instead of proxy statistics. Self-Challenging Language Model Agents (Zhou et al., 2025) let agents propose harder task variants for themselves and then attempt to solve them. Autodata (Kulikov et al., 2026) generalizes the self-instruct line by casting an LLM agent as a meta-optimizing data scientist that iteratively constructs training data; its Agentic Self-Instruct instantiation selects examples by the separation between a strong and a weak reference solver, targeting learnable examples over merely hard ones, a frontier-targeting principle analogous to SPADE's hint-based regret but operating over offline data creation with fixed reference solvers, without the online co-evolution of a gradient-trained generator. Vision-Zero (Wang et al., 2025a) brings gamified multi-agent self-play (Who Is the Spy?) to vision-language models with iterative self-play policy optimization. SPELL (Yang et al., 2026b) co-trains a three-role self-play loop (questioner, responder, verifier) on long documents, using a history-memory curriculum to generate progressively harder questions for evolving long-context capabilities. SPC (Chen et al., 2025b) trains a self-play critic via adversarial games to improve LLM reasoning evaluation. TextArena (Guertler et al., 2025) provides an open-source collection of 57+ competitive text-based game environments with a Gym-compatible API for LLM evaluation and self-play training. Sarkar et al. (2025) train LLMs for Among Us-style social deduction with multi-agent RL. Self-Play with Execution Feedback (Dong et al., 2024) uses code execution to verify instruction-following in a self-play loop. SPIRAL (Liu et al., 2026a) introduces self-play on zero-sum games as a multi-agent, multi-turn RL framework that incentivizes reasoning through game-theoretic competition. SPICE (Liu et al., 2025a) extends self-play to corpus environments via information asymmetry: a Challenger mines documents from a large corpus to generate diverse reasoning tasks with document-grounded answers, while a Reasoner solves them without document access; the corpus prevents hallucination amplification and information-symmetry collapse seen in ungrounded self-play. SWE-RL (Wei et al., 2025a) applies RL to open software evolution tasks, and SSR (Wei et al., 2025b) trains software engineering agents through self-play. Anchored Self-Play (Choi et al., 2026) has a single model play a bug-generator and a fixer for code repair with a difficulty-band reward, adding reference-bug mixing and embedding-similarity shaping to keep self-generated bugs on the realistic distribution, preventing drift off it, and GASP (Jana et al., 2026) guides asymmetric self-play for coding LLMs with grounding in real data. Chae et al. (2025) provide a systematic analysis of when LLM self-play succeeds and when it fails, analyzing the "invisible leash" where generation quality is bounded by base-model capability. Shafayat et al. (2025) investigate whether large reasoning models can self-train and find that naive self-play degrades performance without careful curriculum control.

Liu et al. (2026c) analyze when self-synthetic self-play actually improves, showing that sustained evolution requires the synthetic pipeline to guarantee learnable information gain.

Two themes emerge across this body of work. First, all data-free self-play methods generate *tasks* (a problem statement paired with a sparse terminal reward) rather than full environments with state transitions. Second, the generator in every data-free case is either frozen or trained with heuristic proxy rewards (learnability, difficulty, variance) that risk reward hacking and distributional drift. By contrast, `SPADE` generates *full MDP environments* (state space, transition function, reward function, and verification code) as executable Python, and trains the `Environment Designer` via RL with hint-based regret that is grounded in measured `Reasoning Agent` return rather than proxy statistics. A recent theorem-proving method adds a frozen Guide to score the proposer's conjectures and curb collapse (SGS, Bailey et al., 2026); in `SPADE` the verifier is instead part of each generated environment and co-evolves with the `Environment Designer`.

B.2 Unsupervised Environment Design and Open-Endedness

Curriculum learning (Bengio et al., 2009) established the principle that ordering training examples from easy to hard accelerates learning. Quality-Diversity (QD) algorithms, exemplified by MAP-Elites (Mouret & Clune, 2015), illuminate the search space by maintaining an archive of high-performing solutions across a structured behavior space, providing the algorithmic substrate for many open-ended environment-generation systems. Paired Open-Ended Trailblazer (POET) (Wang et al., 2019) pioneered the paradigm of co-evolving environments alongside agents, using evolutionary search to grow a population of parameterized BipedalWalker terrains together with the policies that solve them, with explicit transfer attempts moving high-performing agents across environments. Unsupervised Environment Design (UED) extends this idea by automatically generating training environments instead of selecting from a fixed pool. PAIRED (Dennis et al., 2020) formalized UED via a minimax regret objective: an adversary generates environment parameters to maximize the gap between an antagonist's and a protagonist's returns, producing curricula of increasing complexity while avoiding unsolvable environments. Dennis et al. (2020) proved that at Nash equilibrium, the protagonist plays a minimax regret policy, connecting UED to decision theory. Prioritized Level Replay (PLR) (Jiang et al., 2021b) replaced the learned adversary with a replay buffer that prioritizes high-regret levels, achieving comparable curriculum quality with lower computational cost. Replay-Guided Adversarial Environment Design (Jiang et al., 2021a) combined the adversarial generator with prioritized replay, using the replay distribution to guide the adversary toward high-regret regions of the environment space. ACCEL (Parker-Holder et al., 2022) introduced evolutionary mutations of high-regret levels, compounding complexity over training without domain-specific heuristics and recovering minimax regret guarantees at a fraction of the compute required by population-based methods like POET. Mediratta et al. (2023) stabilized PAIRED-style learned adversaries via entropy regularization and behavioral cloning between protagonist and antagonist, addressing the entropy collapse that destabilizes UED. Rutherford et al. (2024) investigated regret approximations used in PLR and ACCEL, showing that common proxies (positive value loss, maximum Monte Carlo) correlate with success rate rather than true regret, and proposing Sampling For Learnability (SFL) as an improved estimator. Monette et al. (2025) recast UED as a nonconvex-concave optimization over a categorical level distribution, proving convergence to first-order Nash equilibria and generalizing learnability scores to continuous-return settings. CENIE (Teoh et al., 2024) augmented regret-based UED with a novelty objective that measures how much a candidate environment pushes the agent into unexplored regions of the state-action space, finding that novelty and regret are synergistic rather than competing. DISCOVER (Diaz-Bone et al., 2025) provides a complementary formal-analysis view from goal-conditioned RL: it scores candidate goals via current-policy value estimates balancing achievability, relevance, and novelty, and proves a UCB-style bound on time-to-target-achievability that depends only on the agent's initial distance to the target, independent of task-space volume. The three-axis selection rule parallels `SPADE`'s hint-based regret as a frontier-targeting curriculum signal, but DISCOVER operates over a fixed goal pool whereas `SPADE` generates the environment distribution itself.

A broader thread of open-endedness research motivates the need for unbounded environment spaces. Hughes et al. (2024) argue that open-endedness, defined as simultaneous novelty and learnability, is essential for superhuman AI, and that systems plateau when the environment parameterization is exhausted. Baker et al. (2020) demonstrated emergent tool use from multi-agent autocurricula in hide-and-seek, showing that competitive self-play in rich physics environments produces increasingly

sophisticated strategies. OMNI (Zhang et al., 2023) uses foundation models as a "Model of Interestingness" to filter open-ended task proposals, focusing the curriculum on tasks that are both learnable and interesting. OMNI-EPIC (Faldor et al., 2024) extends this line by representing each environment as executable code generated by a foundation model, with a frozen FM judge of interestingness gating which generated tasks enter the archive; `SPADE` shares the code-as-environment representation but trains the `Environment Designer` via RL with hint-based regret, with no frozen interestingness judge. Imagined Autocurricula (Güzel et al., 2025) applies PLR inside a learned diffusion world model trained from offline data, demonstrating that UED principles transfer to imagined environments. SIMA 2 (Bolton et al., 2025) scales open-ended self-improvement to embodied 3D worlds: a Gemini-based agent uses foundation models to generate its own tasks and rewards and acquires new skills in previously unseen and even model-generated (Genie 3) environments, though it distributes the process across separate task-setter, agent, and reward models plus a distinct generative world model instead of a single policy that authors its own environments. Goldfeder et al. (2026) argue for Superhuman Adaptable Intelligence over AGI, emphasizing speed of adaptation over static benchmark performance. PAPRIKA (Tajwar et al., 2025) trains generally curious agents on ten hand-designed task groups, showing that diverse multi-turn training produces transferable exploration strategies.

All classical UED methods operate in *parameterized* environment spaces (maze dimensions, terrain friction, grid layouts) with small, fixed design vocabularies. To our knowledge, `SPADE` is the first system to bring *regret-based UED with an RL-trained* `Environment Designer` *that produces full MDP environments* to LLM post-training: the environment space is unbounded (arbitrary Python code), the `Environment Designer` *is* the LLM rather than a separate adversary network, and hint-based regret provides the co-evolution signal without requiring a trained antagonist.

B.3 Synthetic Environment Generation

A growing body of work uses LLMs to programmatically generate training environments for agentic RL, addressing the bottleneck of hand-curated environment design. Agent World Model (AWM) (Wang et al., 2026b) decomposes environment synthesis into five stages (scenario, task, database, interface, verification) to produce 1,000 SQLite-backed tool-use environments with MCP interfaces, training agents via GRPO with hybrid step-level and task-level rewards. ScaleEnv (Tu et al., 2026) generates multi-turn tool-use environments by synthesizing API specifications, databases, and verification code from seed domains, scaling to thousands of environments for generalist interactive agent training. TermiGen (Zhu et al., 2026) synthesizes high-fidelity terminal environments inside Docker containers with automated trajectory verification, producing diverse command-line tasks for terminal agent training. Nemotron-Terminal (Pi et al., 2026) introduces Terminal-Task-Gen, a two-stage pipeline that combines *dataset adaptation* (transforming math, code, and SWE benchmarks into terminal-formatted prompts) with *synthetic task generation* (seed-based and skill-based, the latter driven by a Skill Taxonomy of nine domains and primitive skills); the resulting Terminal-Corpus is open-sourced, and Qwen3-32B post-trained via SFT reaches 27.4% on Terminal-Bench 2.0 (from a 3.4% baseline), surpassing Qwen3-Coder-480B (23.9%) at a fraction of the parameter count. SkillSynth (Fan et al., 2026) constructs a scenario-mediated skill graph with 82,073 scenarios as nodes, 57,214 filtered skills as directed transitions, and 185,529 LLM-verified bridges, samples directed paths through the graph as workflow abstractions, and instantiates them via a multi-agent harness with dual execution-based and rubric-based verification (95.7% oracle pass rate, 3,560 verified task instances per run); the explicit objective is maximizing the diversity of execution trajectories over raw task count. Eurekaverse (Liang et al., 2024) uses LLMs to generate Python terrain-program curricula (height-field functions) for robot skill learning, producing progressively harder physics environments guided by agent performance feedback, and validates on quadrupedal parkour. LLM-in-Sandbox (Cheng et al., 2026) places LLMs in sandboxed code-execution environments and shows that agentic intelligence emerges from multi-turn interaction with executable feedback. EvoCUA (Xue et al., 2026a) evolves computer-use agents by synthesizing scalable GUI interaction experiences, generating training trajectories across diverse desktop applications. Xue et al. (2026b) extend this to autonomous continual learning where computer-use agents adapt to new environments through self-generated experience. DreamGym (Chen et al., 2026b) trains an LLM-based world model on demonstration trajectories from existing agentic environments (WebShop, ALFWorld, WebArena) and uses it to generate abstract-text rollouts that replace expensive real-environment interactions during RL. Simia (Li et al., 2025) pushes this direction further by removing real environments entirely: Simia-SFT amplifies small seed trajectories into diverse SFT data via reasoning-model-simulated feed-

back, and Simia-RL performs PPO/GRPO directly against LLM-generated environment transitions; fine-tuned 7–32B Qwen and Llama models reach 36–59 average on τ^2-bench (Airline+Retail subset) without ever executing real tool code, but the framework still inherits the LLM-as-transition-model hallucination risk and the reduced two-domain τ^2 setup is not directly comparable to the three-domain Avg used elsewhere. Lu et al. (2026a) argue that tuning the environment (reward shaping, observation design) can be more effective than tuning the agent, providing a complementary perspective on environment optimization. Exploratory Iteration (Jiang et al., 2025b) grows a self-improvement task space by sampling informative intermediate solution iterates from previous episodes as starting points for new single-step training tasks, training K-step inference-time self-improvement while only training on single-step transitions. Golden Goose (Lu et al., 2026b) synthesizes RLVR tasks from unverifiable internet text by extracting verifiable sub-claims, converting passive corpora into training signal without requiring curated datasets. AutoEnv (Zhang et al., 2025b) provides automated environments for measuring cross-environment agent transfer, enabling systematic evaluation of generalization. Synthetic Sandbox (Zhou et al., 2026) generates sandboxed ML-engineering environments for training software agents on realistic development workflows. RLAnything (Wang et al., 2026a) proposes a dynamic RL system that jointly optimizes policy and reward model while adapting the task distribution via perturbation of existing tasks. GenEnv (Guo et al., 2025) targets difficulty-aligned environment generation with a curriculum reward signal. Agent-World (Dong et al., 2026) mines real-world environments and toolsets from web content (1,978 environments, 19,822 tools) and applies a self-evolving training arena that diagnoses agent weaknesses and generates targeted tasks across rounds. AgentScaler (Fang et al., 2025a) clusters over 30,000 real APIs into more than 1,000 simulated tool-use domains via Louvain community detection on a parameter-similarity graph, materializes each tool as a Python class operating on a per-domain database schema, and trains 4B/8B/30B-A3B agents via two-phase supervised fine-tuning on filtered agent-user trajectories without RL. EnvScaler (Song et al., 2026) programmatically synthesizes tool-interactive environments for multi-turn tool-use agent training, and EnvFactory (Xu et al., 2026b) scales executable tool-use environment synthesis paired with a stable RL recipe. From Trainee to Trainer (Chen et al., 2026a) narrows the adaptivity gap by letting the current RL checkpoint reconfigure its own environment generator from diagnosed failures across training rounds, though it tunes a fixed set of generator parameters and does not emit executable environment code. AutoPlay (Ramrakhya et al., 2026) addresses task synthesis for UI agents (mobile and desktop) by first running an MLLM explorer in each app to gather state and functionality information, then having a frozen GPT-4o task-generator condition on those exploration trajectories plus task-guideline prompts to propose grounded tasks; the executor is trained via SFT and GRPO with binary outcome rewards from an MLLM verifier, gaining 20.6 points on AndroidWorld over Qwen2.5-VL-7B. InfiniteWeb (Zhang et al., 2026e) synthesizes complete static-HTML/localStorage websites alongside tasks and dense-reward JavaScript evaluators via task-centric test-driven development; UI-TARS-1.5-7B post-trained with GRPO on 600 generated tasks improves on OSWorld from 24.5 to 31.4 and on Online-Mind2Web from 23.0 to 28.7, with the largest gains concentrated in easy and medium difficulty buckets. EvoEnv (Shi et al., 2026) co-trains a single LLM under generator and solver role-conditioning via shared-parameter GRPO, producing reusable Python verifier-and-prompt artifacts whose oracle and scorer remain frozen at training time; the generator reward combines staged validation, solver-relative difficulty calibration targeting 30% solver accuracy, and an embedding-based novelty bonus, with evaluation on three model families restricted to single-shot reasoning benchmarks (Qwen3-4B-Thinking-2507 improves from 72.4 to 74.8 averaged across eight math, code, and science benchmarks). The environment interface is single-shot scoring (sampler, oracle, renderer, scorer) without state evolution or step-level rewards, so the framework does not extend to multi-turn agentic settings such as tool use or terminal workflows; `SPADE` differs in three respects: (i) the `Environment Designer` is rewarded via hint-based regret grounded in `Reasoning Agent` return rather than validation and novelty signals, (ii) environments are full MDPs with reset/step interfaces that unify single-turn and multi-turn settings, and (iii) evaluation spans games and tool use in addition to reasoning.

These methods produce useful training material, and some incorporate feedback-driven adaptation where the generation distribution shifts in response to agent progress: Eurekaverse evolves environments based on agent performance statistics, and EvoCUA iterates synthesis with an improving policy. However, even in these cases the generator LLM receives no RL gradients; adaptation operates through prompting heuristics or iterative data filtering, without joint optimization. By contrast, `SPADE`'s `Environment Designer` is a genuine RL learner, trained via hint-based regret in

the same optimization loop as the `Reasoning Agent`, producing full MDP environments at the `Reasoning Agent`'s competence frontier.

Further systems add dynamic RL pipelines (Wang et al., 2026a; Guo et al., 2025; Shi et al., 2026) and computer-use sandboxes (Fan et al., 2026; Xue et al., 2026b; Zhang et al., 2026e), with a robotics analogue in Eurekaverse (Liang et al., 2024). Complementary task-level synthesis grounds generation in agent exploration (Ramrakhya et al., 2026), converts unverifiable text into RLVR data (Lu et al., 2026b), or expands seed task spaces (Jiang et al., 2025b; Chen et al., 2026b; Li et al., 2025; Lu et al., 2026a).

B.4 Environment Scaling

Following the shift from single-turn reinforcement learning with verifiable rewards (RLVR) (OpenAI, 2024; DeepSeek-AI, 2025) to multi-turn agentic RL (Wang et al., 2025c; Zhang et al., 2025c), the environment pool has become the bottleneck. A growing consensus holds that the next frontier for agentic RL is scaling environments rather than algorithms. Silver & Sutton (2025) articulate this vision most directly, arguing that the field is entering an "era of experience" in which the primary bottleneck is generating sufficiently diverse and abundant training environments. Zhang et al. (2026b) argue in the same vein that scalable, diverse environments are what drive generalizable agents. AgentRL (Zhang et al., 2025a) provides a multi-turn, multi-task RL framework with a fully asynchronous generation-training pipeline, cross-policy sampling for improved exploration, and task advantage normalization, training a single agent across five agentic domains to outperform frontier models. SCALER (Xu et al., 2026a) converts competitive programming problems (from CodeContests) into parameterized single-turn reasoning environments at controlled difficulty levels, demonstrating that scaling the number and diversity of training problems yields consistent improvements in mathematical reasoning. RLVE (Zeng et al., 2025) creates 400 hand-engineered verifiable environments for RL training with adaptive difficulty levels and per-environment custom verifiers (mixing rule-based parsing, soft scoring, and code execution) in place of LLM-as-judge. WebScale-RL (Cen et al., 2026) builds an automated data pipeline that harvests web content to create RL training environments at pretraining scale, showing that more diverse environments produce better generalization. Endless Terminals (Gandhi et al., 2026) procedurally generates 3,255 verified terminal-use tasks via a four-stage pipeline and demonstrates that vanilla PPO with binary rewards yields substantial gains as the number of training environments scales, with transfer to TerminalBench 2.0. Self-Evolving Curriculum (Chen et al., 2025d) dynamically adjusts the difficulty distribution of training problems based on the learner's current performance, preventing saturation on easy examples. TTCS (Yang et al., 2026a) co-evolves a gradient-trained problem synthesizer with a solver via GRPO, using a capability-adaptive reward that targets the solver's learning frontier at test time. Xu et al. (2025) survey the path toward Large Reasoning Models, organizing the field into three pillars (automated data construction, learning-to-reason via RL, and test-time scaling) and identifying environment scaling as a key open problem. Song et al. (2024) examine self-improvement capabilities of LLMs and find that naive self-training saturates quickly, motivating the need for adaptive curricula. Embodied Co-Design (Wang et al., 2025b) provides a taxonomy of co-design approaches where agent morphology and controller evolve jointly, drawing parallels to `SPADE`'s co-evolution of environment and agent. Agent Learning via Early Experience (Zhang et al., 2025c) introduces a reward-free training paradigm in which agents propose alternative actions at expert-visited states and learn from the resulting future states (implicit world modeling and self-reflection), bridging imitation learning and full RL across eight agentic environments.

These works independently demonstrate that more environments yield better generalization and that curriculum design matters at least as much as algorithm choice. Several incorporate adaptive mechanisms: RLVE adjusts per-environment difficulty levels, SCALER tracks accuracy for difficulty control, and Self-Evolving Curriculum learns a sampling policy over problem categories. `SPADE` combines gradient-trained environment design with the code-as-environment representation, producing an adaptive curriculum of executable MDPs that co-evolves with the `Reasoning Agent`.

B.5 Agentic Memory Design and Self-Improving Code Systems

MemRL (Zhang et al., 2026d) introduces self-evolving agents that learn via runtime reinforcement learning on episodic memory, storing and retrieving past interaction experiences to improve future decision-making without additional gradient updates. ALMA (Xiong et al., 2026) meta-learns agentic memory designs themselves: a foundation model searches the space of Python memory-architecture programs (an open-ended scaffold over update/retrieve operations), discovering memory designs that improve continual learning at test time. HyperAgents (Zhang et al., 2026a) extends self-referential code evolution by fusing the task-solver and the meta-improver into a single editable program, so the self-modification mechanism is itself modifiable; tasks are externally fixed and the foundation model is frozen. These directions are complementary to SPADE: they address how agents *use* accumulated experience or *rewrite their own scaffolding* at the meta-level, while SPADE addresses how to *generate* the training environments that produce that experience in the first place via RL. HyperAgents explicitly leaves co-evolving the task distribution as future work, which SPADE addresses directly. The approaches are composable: SPADE's adaptive environment generation could populate the experience streams that ALMA-style memory systems and MemRL-style runtime RL operate over.

C Notation

Table 5 consolidates symbols used throughout the paper.

Table 5: Symbols used throughout the paper.

Symbol	Description
MDP and environment	
$\mathcal{S}$	State space
$\mathcal{A}$	Action space
$T(s' \mid s, a)$	Transition function
$R(s, a)$	Reward function
ρ_0	Initial state distribution
s, s', a	State, next state, action
$\mathcal{E}$	Space of valid (executable Python) environments
$e \in \mathcal{E}$	A single environment instance
e_b	The b-th environment in a generation batch
Policy and roles	
π_θ	Shared LLM policy with parameters θ
π_D	Policy in `Environment Designer` role, $\pi_\theta(\cdot \mid \text{role}{=}D)$
π_A	Policy in `Reasoning Agent` role, $\pi_\theta(\cdot \mid \text{role}{=}A)$
role$=D$	System-prompt switch selecting `Environment Designer` role
role$=A$	System-prompt switch selecting `Reasoning Agent` role
Hints and rewards	
h	Privileged hint (strategy / partial solution / key observation)
h_b	Hint for the b-th environment
y, y_i	`Reasoning Agent` response (rollout)
y'_i	`Reasoning Agent` rollout sampled with the privileged hint in context
$r_A(y \mid e)$	Per-rollout correctness reward (without hint)
$r_A(y \mid e, h)$	Per-rollout correctness reward conditioned on hint
$\bar{r}_A(e)$	Average `Reasoning Agent` return on e without hints
$\bar{r}_A(e \mid h)$	Average `Reasoning Agent` return on e with hint h
$r_D(e)$	`Environment Designer` reward: hint-based regret $\bar{r}_A(e \mid h) - \bar{r}_A(e)$
GRPO and training	
x	Prompt / input sequence (generic GRPO notation)
$\mathcal{L}(\theta)$	GRPO clipped-surrogate training objective
$\pi_{\text{old}}, \pi_{\text{ref}}$	Behavior policy (importance ratio) and KL reference policy
$\hat{A}^i$	Group-normalized advantage (generic); role-specific forms below
$\hat{A}^b_D$	Mean-centered advantage within its skill group (no standard-deviation scaling) for the b-th designed environment
$\hat{A}^i_A$	Group-normalized advantage for the i-th agent rollout
$\varepsilon_{\text{low}}, \varepsilon_{\text{high}}$	Asymmetric PPO clipping range (DAPO clip-higher (Yu et al., 2025))
β_{KL}	KL-regularization coefficient
B	Generation batch size (environments per iteration)
G	Group size (`Reasoning Agent` rollouts per environment)
k	Regeneration interval: rollouts per environment set, and the `Environment Designer` update delay
N	Total number of training iterations
p_d	Domain prompt sampled to seed environment generation
r^i, r^j	Scalar reward of the i-th / j-th response in a GRPO group
M	Environment memory: buffer of past environments with regret scores and skill tags
C	Pretraining corpus used to ground environment design

D Theoretical Analysis

This section formally analyzes the incentive structure of the hint-based regret reward (hereafter hint-regret) introduced in Section 4.2. We model the interaction between the `Environment Designer` and `Reasoning Agent` as a two-player game: the `Environment Designer` selects an environment distribution, while the `Reasoning Agent` selects a policy. Since LLM inference is stochastic, all payoffs are defined through expected verifier returns.

The main result is an equilibrium characterization of the idealized game. If positive hint-regret remains anywhere, then the current unhinted `Reasoning Agent` is suboptimal there, and the `Environment Designer` can profitably concentrate on it. Consequently, at any pure Nash equilibrium, the `Environment Designer`'s expected regret is zero, the `Reasoning Agent` is hint-free optimal on *every* environment in $\mathcal{M}$, and privileged hints become vacuous.

D.1 Setup and Assumptions

Environment class. Let $\mathcal{U}$ be a finite universe of candidate environments. The set of *mathematically valid executable* environments is denoted by $\mathcal{M} \subseteq \mathcal{U}$. These are environments whose specifications are internally consistent and whose attempted solutions can be evaluated by an external verifier. In the implementation, $\mathcal{M}$ is approximated by syntax checks, executability checks, and, in the tool-use setting, solvability filtering.

Distribution notation. For any finite set $\mathcal{X}$, let $\Delta(\mathcal{X})$ denote the set of probability distributions over $\mathcal{X}$. If $D \in \Delta(\mathcal{M})$ (the `Environment Designer`'s environment distribution; we reuse the letter D for this distribution, while the subscript in u_D below labels the `Environment Designer` role) and $\mathcal{B} \subseteq \mathcal{M}$, write

$$D(\mathcal{B}) := \sum_{e \in \mathcal{B}} D(e).$$

A property holds for D-almost every environment if the set on which it fails has D-mass zero.

Inference protocol. Fix an inference budget B_{inf}. This budget includes all evaluation-time choices that affect the set of possible `Reasoning Agent` trajectories: model architecture, context length, prompt format, maximum generation length, sampling rule, temperature, number of rollouts, tool access, memory access, and verifier. Let $\mathcal{Y}_{B_{\text{inf}}}$ denote the finite trajectory space induced by this budget.

Fine-tuning budget. Fix a fine-tuning budget F. This budget includes all training-time choices that determine which policies the procedure can produce: optimizer, objective, number of updates, and regularization. Let Π denote the set of policies attainable under this budget; we take Π to be finite, so maxima over Π are attained.

Returns with and without hints. For each environment $e \in \mathcal{M}$, let $h(e)$ denote the privileged hint emitted by the `Environment Designer`; hints are modeled as a fixed map $e \mapsto h(e)$, so strategic hint choice by the `Environment Designer` is outside this idealization. A `Reasoning Agent` policy $\pi \in \Pi$ induces a distribution over trajectories in $\mathcal{Y}_{B_{\text{inf}}}$ given either the unhinted input e or the hinted input $(e, h(e))$. Let

$$r(e, y) \in [0, 1]$$

denote the verifier return of trajectory $y \in \mathcal{Y}_{B_{\text{inf}}}$ on environment e. Returns are normalized to $[0, 1]$ for the analysis; training uses a shaped reward in $[-1, 1]$ (Algorithm 1). Here $r(e, y)$ is the per-trajectory return written $r_A(y \mid e)$ in Section 4, and $R_\pi(e)$, $R^h_\pi(e)$ below are the population analogues of the empirical means $\bar{r}_A(e)$ and $\bar{r}_A(e \mid h)$.

The unhinted expected return of policy π on environment e is

$$R_\pi(e) := \mathbb{E}_{y \sim \pi(\cdot|e)}[r(e, y)],$$

and the hinted expected return is

$$R^h_\pi(e) := \mathbb{E}_{y \sim \pi(\cdot|e,h(e))}[r(e, y)].$$

Game payoffs. The `Environment Designer` chooses a distribution $D \in \Delta(\mathcal{M})$. Its payoff is the expected hint-based regret

$$u_D(D, \pi) := \mathbb{E}_{e \sim D}\left[R^h_\pi(e) - R_\pi(e)\right].$$

The `Reasoning Agent`'s payoff is its unhinted expected return

$$u_A(D, \pi) := \mathbb{E}_{e \sim D}[R_\pi(e)].$$

Together these define the expected-regret self-play game on $\Delta(\mathcal{M}) \times \Pi$.

Pure Nash equilibrium. A pair $(D^\circ, \pi^\circ) \in \Delta(\mathcal{M}) \times \Pi$ is a pure Nash equilibrium of this game if neither player can improve by unilateral deviation:

$$u_D(D^\circ, \pi^\circ) \geq u_D(D, \pi^\circ) \qquad \forall D \in \Delta(\mathcal{M}),$$

and

$$u_A(D^\circ, \pi^\circ) \geq u_A(D^\circ, \pi) \qquad \forall \pi \in \Pi.$$

Optimal values. For each environment, define the optimal unhinted value

$$R^\star(e) := \max_{\pi \in \Pi} R_\pi(e).$$

Assumption D.1 (Sound generation)**.** The `Environment Designer` only samples mathematically valid executable environments:

$$D \in \Delta(\mathcal{M}).$$

Operationally, this corresponds to syntax checks, executability checks, and, in the tool-use setting, solvability filtering. (Stated for completeness: the game is defined over $\Delta(\mathcal{M})$ throughout.)

Assumption D.2 (Articulated hints)**.** Conditioning on the hint attains the optimal unhinted value: for every policy $\pi \in \Pi$ and every environment $e \in \mathcal{M}$,

$$R^h_\pi(e) = R^\star(e).$$

In particular $R^h_\pi(e) \geq R_\pi(e)$.

Assumption D.3 (Internalizability of hinted behavior)**.** Hinted behavior is attainable without the hint: for every policy $\pi \in \Pi$ there exists a policy $\pi' \in \Pi$ with

$$R_{\pi'}(e) = R^h_\pi(e) \qquad \text{for all } e \in \mathcal{M}.$$

D.2 Main Results

Theorem D.5 shows that every pure Nash equilibrium in this setup drives hint-regret to zero, so that the `Reasoning Agent` is optimal on every environment in $\mathcal{M}$. The ideal hint-regret enables this result, because it vanishes on environments the `Reasoning Agent` already solves and on unsolvable environments, where the hint cannot help, and is positive only where the `Reasoning Agent` fails unaided but a hint would close the gap. The `Environment Designer` therefore profits exactly by targeting this learnable frontier, and at equilibrium no gap can remain anywhere, since the `Environment Designer` could otherwise deviate to it profitably. This is what distinguishes the regret reward from the plain adversarial-difficulty reward $-\mathbb{E}_{e\sim D}[R_\pi(e)]$, which provides no mechanism restricting the `Environment Designer`'s support to useful environments.

Lemma D.4 (Hint-regret equals hint-free regret)**.** *Under Assumption D.2, for any* `Reasoning Agent` *policy* $\pi \in \Pi$ *and any environment* $e \in \mathcal{M}$*,*

$$R^h_\pi(e) - R_\pi(e) = R^\star(e) - R_\pi(e).$$

In particular the hint-regret is nonnegative, and strictly positive if and only if $R_\pi(e) < R^\star(e)$.

Proof. By Assumption D.2, $R^h_\pi(e) = R^\star(e)$; subtracting $R_\pi(e)$ gives the identity. The remaining claims follow since $R^\star(e) \geq R_\pi(e)$. □

Theorem D.5 (Nash equilibria imply hint-free optimality on every environment)**.** *Under Assumptions D.1–D.3, every pure Nash equilibrium* (D°, π°) *of the expected-regret self-play game satisfies*

$$u_D(D^\circ, \pi^\circ) = 0.$$

Moreover, the pointwise regret vanishes everywhere: for every $e \in \mathcal{M}$*,*

$$R_{\pi^\circ}(e) = R^\star(e), \qquad R^h_{\pi^\circ}(e) = R_{\pi^\circ}(e).$$

Such equilibria exist: Assumptions D.2 and D.3 guarantee a uniformly optimal policy $\pi^\star$ with $R_{\pi^\star}(e) = R^\star(e)$ for all e, and any pair $(D, \pi^\star)$ is a pure Nash equilibrium, so the statement is not vacuous.

Proof. Let (D°, π°) be a pure Nash equilibrium. By Lemma D.4,

$$\rho^{\text{reg}}(e) := R^h_{\pi^\circ}(e) - R_{\pi^\circ}(e) = R^\star(e) - R_{\pi^\circ}(e) \geq 0 \qquad \text{for every } e \in \mathcal{M}.$$

`Reasoning Agent` side. By Assumption D.3, there is $\pi' \in \Pi$ with $R_{\pi'}(e) = R^h_{\pi^\circ}(e)$ for all e; by Assumption D.2, $R^h_{\pi^\circ}(e) = R^\star(e)$, so $R_{\pi'}(e) = R^\star(e) \geq R_{\pi^\circ}(e)$ for every $e \in \mathcal{M}$. Hence

$$u_A(D^\circ, \pi') = \mathbb{E}_{e\sim D^\circ}[R^\star(e)] \geq \mathbb{E}_{e\sim D^\circ}[R_{\pi^\circ}(e)] = u_A(D^\circ, \pi^\circ).$$

Since π° is a best response to D°, equality holds, so $\mathbb{E}_{e\sim D^\circ}[R^\star(e) - R_{\pi^\circ}(e)] = 0$. The integrand is nonnegative, hence $\rho^{\text{reg}}(e) = 0$ for D°-almost every e, and therefore

$$u_D(D^\circ, \pi^\circ) = \mathbb{E}_{e\sim D^\circ}[\rho^{\text{reg}}(e)] = 0.$$

`Environment Designer` side. Fix any $e \in \mathcal{M}$. The `Environment Designer` may deviate to the point mass $\delta_e \in \Delta(\mathcal{M})$, which yields $u_D(\delta_e, \pi^\circ) = \rho^{\text{reg}}(e)$. Since (D°, π°) is a Nash equilibrium,

$$\rho^{\text{reg}}(e) = u_D(\delta_e, \pi^\circ) \le u_D(D^\circ, \pi^\circ) = 0.$$

Combined with $\rho^{\text{reg}}(e) \ge 0$ from Lemma D.4, this gives $\rho^{\text{reg}}(e) = 0$ for *every* $e \in \mathcal{M}$. Therefore

$$R_{\pi^\circ}(e) = R^\star(e), \qquad R^h_{\pi^\circ}(e) = R_{\pi^\circ}(e)$$

for every $e \in \mathcal{M}$. □

E Method and Experiment Details

E.1 GRPO Objective

Each role's payoff is optimized with reinforcement learning from verifiable rewards (RLVR) using Group Relative Policy Optimization (GRPO) (Shao et al., 2024). For an input prompt x (a role prompt together with its environment, and the hint for hinted rollouts), GRPO samples a group of G responses $\{y_1, \dots, y_G\} \sim \pi_\theta(\cdot \mid x)$ and computes group-normalized advantages:

$$\hat{A}^i = \frac{r^i - \text{mean}(\{r^j\}_{j=1}^G)}{\text{std}(\{r^j\}_{j=1}^G)} \tag{3}$$

Define the importance ratio $w_i(\theta) = \pi_\theta(y_i \mid x)/\pi_{\text{old}}(y_i \mid x)$. The policy is updated via clipped policy gradient with KL regularization:

$$\begin{aligned}\mathcal{L}(\theta) = &-\frac{1}{G}\sum_{i=1}^{G} \min\Big(w_i(\theta)\hat{A}^i, \text{clip}(w_i(\theta), 1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}})\,\hat{A}^i\Big) \\ &+ \beta_{\text{KL}}\,\text{KL}[\pi_\theta \| \pi_{\text{ref}}].\end{aligned} \tag{4}$$

Advantages are normalized per role as in Algorithm 1: `Reasoning Agent` returns are standardized within each environment's rollouts, and `Environment Designer` rewards are mean-centered within each skill.

Terminal tasks. Following Tmax (Ivison et al., 2026), the terminal runs replace the clipped surrogate with DPPO (Qi et al., 2026), which keeps the unclipped surrogate and bounds each update with a direct estimate of the per-token policy divergence. With μ_t and p_t the probabilities that π_{old} and π_θ assign to the sampled token, the total-variation divergence of the binary distribution {sampled token, all other tokens} is $|\mu_t - p_t|$. A token is masked out of the policy gradient when this divergence exceeds 0.1 and the update would move its ratio further from 1. There is no KL or entropy term (Table 7), and advantages use the same per-role normalization.

E.2 Implementation Details

Two auxiliary `Environment Designer` reward terms are disabled in all full-`SPADE` runs: a per-environment EMA-based learning-potential bonus (Kanitscheider et al., 2021; Zhang et al., 2023), enabled only as the standalone reward replacement evaluated in Section 6.3, and a frontier bonus (extra reward for environments with a large fast-versus-slow EMA gap $|\mu_{\text{fast}} - \mu_{\text{slow}}|$). No separate variance bonus (an $\bar{r}_A(1-\bar{r}_A)$-style reward for mixed outcomes) is used. Failed environment candidates are regenerated up to a fixed number of attempts. Table 6 lists the full training configuration.

EMA-based learning-potential reward (ablation detail). The standalone learning-potential reward evaluated in Section 6.3 scores an environment by how far the `Reasoning Agent`'s success on it departs from its skill's recent average. Environments are grouped by skill s. Let $\bar{r}_A(e)$ be

the `Reasoning Agent`'s average return on environment e (Eq. 2), and $\bar{r}_{A,t}(s)$ its mean over the skill's environments in scoring round t. Two moving averages of $\bar{r}_{A,t}(s)$ are tracked at rates $\gamma_1 > \gamma_2$:

$$\mu_t^{\gamma}(s) \;=\; (1-\gamma)\,\mu_{t-1}^{\gamma}(s) \;+\; \gamma\,\bar{r}_{A,t}(s), \qquad \mu_{\text{fast}} \equiv \mu^{\gamma_1}, \quad \mu_{\text{slow}} \equiv \mu^{\gamma_2}. \tag{5}$$

Here γ weights the new observation, so μ_{fast} adapts faster than μ_{slow}, and μ^{γ} has half-life $\log(0.5)/\log(1-\gamma)$ rounds. The reward uses μ_{slow} as the skill baseline:

$$\rho(e) \;=\; \left|\, \bar{r}_A(e) - \mu_{\text{slow},t}(s) \,\right|, \tag{6}$$

$$r_D^{\text{LP}}(e) \;=\; \rho(e) - \tfrac{1}{|\mathcal{E}_s|} \textstyle\sum_{e' \in \mathcal{E}_s} \rho(e'), \tag{7}$$

where $\mathcal{E}_s$ is the round's environment set for skill s. The deviation is unsigned, so an environment the `Reasoning Agent` always solves can score as highly as one it never solves when the slow mean sits mid-range, and neither gives gradient. The signal also needs a per-skill history before it is meaningful, and cannot separate deviation caused by environment design from `Reasoning Agent` drift or sampling noise.

Pool lifecycle. At each regeneration the previous environment set is deleted and replaced; the environment memory evicts oldest-first at its 200-record cap; rejected candidates are persisted for inspection; generation retries up to five attempts per environment.

Hint generation detail. The privileged hint is produced by a separate designer-side call conditioned on the generated environment's source code: the hint writer sees the code, while the `Reasoning Agent` never does.

Validation detail. Every candidate environment must pass a programmatic smoke test: the class is instantiated, `reset()` is called, and a few probe actions are stepped through, discarding candidates that fail to parse or crash. The tool-use setting adds the two semantic checks of Section 5.2: a deterministic reset gate that rejects environments in which a success criterion errors on the freshly reset state under every tested seed, and an LLM screen that rejects impossible environments (unreachable, pre-satisfied, or underivable success criteria) while keeping hard-but-feasible ones.

E.2.1 Terminal Training Details

The comparison holds the agent-side training settings fixed across all three conditions. Each rollout iteration targets 16 nondegenerate task groups, each with 16 episodes. A fixed oversampling draw admits up to 28 candidate groups, and groups with constant rewards are dropped. Partial-rollout carry preserves unfinished episodes across iteration boundaries.

For the two online `SPADE` variants, a generation phase runs every eight recorded training iterations and targets 48 tasks sampled uniformly across six terminal skills. For each candidate, the current checkpoint generates the public task and private truth, the initial-state verifier, and the final-state verifier before materializing the container setup and oracle; only artifacts that build and pass validation enter the pool. After validation, a sixth call writes the hint used for regret scoring. Each phase allows at most 300 attempts and one repair pass. If fewer than eight new tasks survive, validated tasks from the previous phase are reused. Full `SPADE` additionally applies explicit `Environment Designer` policy-gradient updates to the environment-design trajectories using hint-based regret; the No Proposer Training arm excludes those trajectories from optimization. For terminal, we switch to **miles** (Team, 2026) as our RL framework, given its better support on Nemotron models.

E.2.2 Online Terminal Environment Construction

We adapt Tmax's compositional task-generation recipe (Ivison et al., 2026), which controls task diversity and difficulty through nine axes: domain, skill type, primitive skills, persona, language, task complexity, command complexity, fixture kind, and verifier kind. Our study samples a target uniformly from six terminal skill categories, then samples its primitive skills, implementation language, and difficulty; it fixes the fixture and verifier kinds to text-only and exact-text, respectively, and uses a fixed Python 3.12 base image. Tmax describes a compositional generator followed by an executability filter. Our six-stage pipeline, five artifact-construction calls followed by hint writing, together with its trainable stage trajectories and oracle-based validation, is a `SPADE`-Terminal extension rather than a sequence claimed by Tmax.

Table 6: **Training hyperparameters.** Shared across the three games-setting SPADE backbone runs and the tool-use runs; per-model and tool-use exceptions appear in parentheses.

Hyperparameter	Value
Models	Qwen3-4B-Instruct-2507, Qwen3-8B, Qwen3-30B-A3B-Instruct-2507
Learning rate	1×10^{-6} (constant)
Optimizer	Adam, $\beta = (0.9, 0.98)$, weight decay 0.1
KL penalty β_{KL}	0 (0.005 for 8B)
Clipping $\varepsilon_{\text{low}}/\varepsilon_{\text{high}}$	$0.20/0.28$
Truncated importance sampling	yes
Reward normalization	outcome-only, per-game z-score
Rollout batch	24
Global batch	192 (dynamic)
Group size G	16
Total rollouts	400
Environments per rollout	24 (8×3 active skills of 6, round-robin)
Regeneration interval k	4 rollouts (8 for tool use)
`Environment Designer` temperature	0.6
`Environment Designer` max tokens	$16{,}384$ ($20{,}000$ for 8B)
`Reasoning Agent` temperature	0.6
`Reasoning Agent` max tokens	$8{,}192$
Max turns per episode	25
Max context length	$32{,}768$ ($49{,}152$ later in the 4B run)
`Environment Designer` reward blend	plateau 0.6, band $[0.4, 0.6]$ ($[0.2, 0.4]$ later for 4B) $+$ floored regret 0.4
Regret scale (normalizer)	0.15
Plateau ramp width	0.25
Delayed `Environment Designer` update	4 rollouts (8 for tool use)
Corpus grounding	15k docs (10k math, 5k science; 15k code docs for tool use)
Environment memory	on

Table 7: **Terminal training configuration.** Agent-side settings are held fixed across the three conditions; online-generation settings apply to the two SPADE variants.

Hyperparameter	Value
Base model	NVIDIA-Nemotron-3.5-Lightning-30B-A3B-BF16
Fine-tuning	Full-parameter
Objective	DPPO; TV divergence threshold 0.1
Training budget	300 rollout iterations; checkpoint every 8 iterations
Learning rate	10^{-6} (constant)
Optimizer	Adam, $\beta = (0.9, 0.98)$; weight decay 0.1; gradient clip 1.0
KL / entropy coefficients	0 / 0
Agent batch target	16 tasks $\times$ 16 episodes
Group collection	Up to 28 candidate groups per iteration; zero-variance groups removed
Partial-rollout carry	Enabled across iteration boundaries
Sampling	Temperature 1.0; top-p 1.0; top-k disabled
Episode limits	128 turns; 3,600 seconds
Token limits	16,384 response; 98,304 context
Online generation phase	Every 8 recorded iterations; target 48 validated tasks; 6 skills, sampled uniformly

Table 8: **Terminal environment construction and trainability.** "Trainable" means that the call contributes an `Environment Designer` trajectory in full `SPADE`; the No Proposer Training arm executes the same pipeline but excludes these trajectories from optimization. Stage 6 occurs only after the first five stages pass validation.

Phase	Call	Purpose and output	Trainable in full SPADE
1	Task design	Writes the public task description and private truth, conditioned on the sampled skill, primitives, language, and difficulty.	Yes
2	Initial verifier	Writes `test_initial_state.py`, which checks required fixtures and preconditions before the actor acts.	Yes
3	Final verifier	Writes `test_final_state.py`, which encodes successful completion without adding requirements absent from the public task.	Yes
4	Setup materializer	Writes `setup.sh` and deterministic fixtures that create, but do not solve, the initial sandbox state.	No
5	Oracle materializer	Writes `oracle.sh`, a reference solution used to establish that the final verifier can be satisfied.	No
6	Hint writing	After validation, uses the public task alone to write a lossy orientation plan for the hinted actor rollouts used to estimate regret.	No

For phases 1–3, full `SPADE` retains the first-attempt prompt tokens, output tokens, and rollout log-probabilities and converts each phase into a separate `Environment Designer` training trajectory. The task-level environment reward is split across the three trajectories so that one generated task preserves the gradient weight of one environment sample. Inference-only repairs may make an artifact usable, but repair tokens never replace the recorded first attempt; setup, oracle, and hint tokens never enter the training batch.

E.2.3 Executable Validation and Generation Yield

We define the usable generation rate over candidate attempts before pool reuse as

$$\text{usable rate} = \frac{N_{\text{usable}}}{N_{\text{attempted}}}. \tag{8}$$

A candidate is usable only if it passes structural parsing and all four executable gates: (i) the setup completes in the target sandbox; (ii) the initial-state verifier passes before the actor acts; (iii) the final-state verifier fails in that untouched state, ruling out a pre-solved task; and (iv) the oracle completes successfully and makes the final-state verifier pass. This is deliberately stronger than Tmax's build/executability filter, which omits expensive teacher-based correctness validation and relies on zero-pass-rate filtering during RL (Ivison et al., 2026).

Table 9: **Usable terminal-environment generation rate by model.** Rates are rounded from the internal generator screening run and apply the artifact and executable gates above.

Generator	Usable rate
`inkling-small`	52%
Qwen3.6-35B-A3B	44%
DeepSeek-V4-Flash-0731	39%
Nemotron-3.5-Lightning	27%
Qwen3.5-9B	9%
Nemotron-3-Nano	5%

These rates measure whether a model can construct a training environment, not whether that model is the strongest terminal actor.

E.2.4 Terminal Hint Generation and Regret

Hint generation runs after a candidate passes the executable gate. The hint writer receives only the public task description, the same information available to the actor, and never sees the private truth, setup script, verifiers, or oracle. It produces a two- or three-sentence, non-copyable orientation plan: the broad phases of the task, a suggestion to inspect relevant inputs before modifying state,

and a reminder to validate the result before submitting. Exact command sequences and undisclosed expected outputs are prohibited.

Both hinted and unhinted actor plays are inference-only. Their return gap supplies the hint-based regret of Eq. 2, and the resulting task-level reward is assigned to the trainable task-design and verifier trajectories in phases 1–3. The hint is therefore a downstream probe of the task's learning potential, not an additional environment artifact and not itself a trained output.

E.3 Reproducibility

The training and evaluation code is available at the anonymous repository linked in the abstract, including launch scripts for the Qwen3 games, tool-use, and ablation runs, the terminal runs, and the offline evaluation harness.

E.3.1 Comparability Notes for Table 2

The reference systems in Table 2 were trained and evaluated by their own authors on their own harnesses; we reprint their published scores for context and record the protocol differences here. **BFCL versions.** Our rows and Agent-World (Dong et al., 2026) report the BFCL v4 multi-turn suite. AWM (Wang et al., 2026b) and EnvScaler (Song et al., 2026) label the suite v3, which contains the same four multi-turn subcategories; EnvScaler evaluates a frozen 2024-09-22 data snapshot whose repository notes minor differences from later releases. **τ^2-bench variants.** AWM evaluates τ^2-bench-verified, a fork with task corrections, runs the Telecom domain in solo mode (agent only, no user simulator), and uses a GPT-5.1 user simulator in the other domains; AgentScaler (Fang et al., 2025a) does not state its user simulator. We evaluate the standard τ^2-bench release. **Aggregation.** Reference-row Avg cells follow the rule stated in the table caption; Agent-World and AWM print their own aggregates (61.8/65.4 and task-weighted 33.5/39.0), which do not equal the unweighted means of the domain scores they report. **Base models.** Our rows post-train Qwen3-4B-Instruct-2507, Qwen3-8B, and Qwen3-30B-A3B-Instruct-2507; the reference systems post-train their own base checkpoints, with training data and budgets that differ from ours.

F Extended Results

Throughout, environment text is embedded with SBERT (`all-MiniLM-L6-v2`) (Reimers & Gurevych, 2019), and every embedding-based result is replicated under a second, purely lexical embedding (TF-IDF with LSA-128), following the multi-embedding robustness protocol of Abdulhai et al. (2026); conclusions are identical under both. Step-level training dynamics use the matched 30B-A3B runs (full `SPADE` is the resume-stitched 0–399 step run; ablations end earlier), with curves EMA-smoothed over real logged values, expanding on Section 6.2.

The games-setting analyses below expand the main-text results and the environment-quality and diversity signals.

F.1 Main-Result Trajectories

Figure 7 plots the held-out benchmark trajectories over training behind the Table 1 results for the primary 30B-A3B games run.

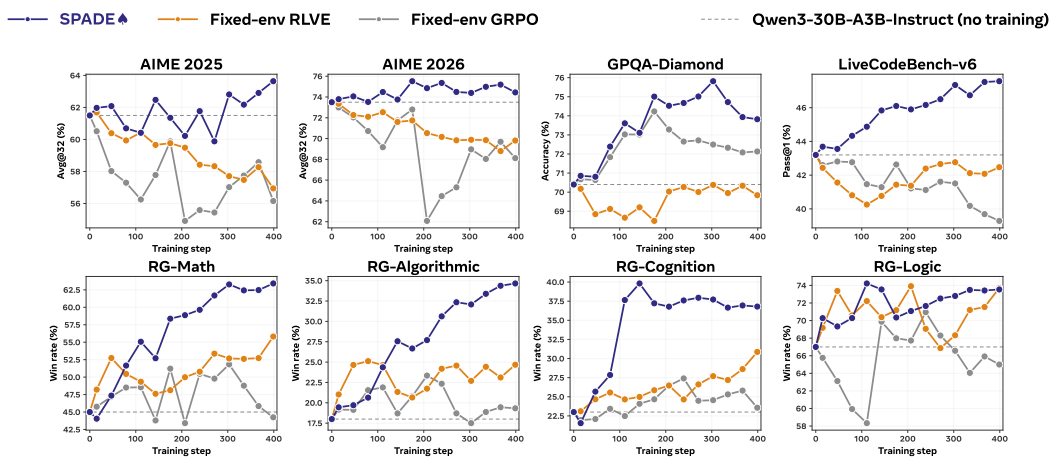


Figure 7: **Training on diverse synthetic games improves science reasoning, code generation, and procedural reasoning while competition math is preserved** (games setting, Qwen3-30B-A3B-Instruct-2507). Top row: competition math (AIME 2025/2026 Avg@32), science reasoning (GPQA-Diamond accuracy), and code generation (LiveCodeBench-v6 Pass@1). Bottom row: procedural reasoning across four cognitive skills (Reasoning-Gym win rate at HARD). Markers denote logged evaluation checkpoints; the dashed line marks the untrained base model.

F.2 Component Ablations

Table 10 gives the per-benchmark results behind the ablation summary of Table 4.

Table 10: **The full adaptive configuration outperforms every partial and non-adaptive control.** Games, Qwen3-30B-A3B-Instruct-2507; best checkpoint per variant on suite average. Avg is the unweighted mean over the eight benchmarks of Table 1 (full trajectories in Figure 8). Best in **bold**, second underlined.

	Components				Benchmarks								
Setting	ED design	ED trained	Corpus grounding	Env. memory	AIME'25	AIME'26	GPQA-D	LCB-v6	RG-Math	RG-Algo.	RG-Cog.	RG-Logic	**Avg**
Qwen3-30B-A3B-Instruct-2507	–	–	–	–	61.5	73.5	70.4	43.2	45.0	18.0	23.0	67.0	50.2
SPADE♠	Self	✓	✓	✓	**62.8**	74.4	**75.8**	**47.3**	**63.3**	**32.1**	**37.7**	**72.8**	**58.3**
w/o memory	Self	✓	✓	✗	59.3	**75.0**	72.3	45.7	49.1	22.9	30.7	70.9	53.2
w/o corpus grounding	Self	✓	✗	✓	61.1	74.1	71.8	46.3	51.6	22.3	32.4	68.7	53.5
w/o ED training and memory	Self	✗	✓	✗	59.4	73.5	65.8	39.1	22.5	10.0	7.6	46.0	40.5
Fixed `Environment Designer` (GPT-5.5)	GPT-5.5	✗	✓	✗	59.9	72.8	74.2	42.6	51.2	24.3	30.7	68.0	53.0

Table 11 expands the component-controlled ablations of Table 4 to every games-setting variant reported in this paper, including the skill-diversity run, with each variant's selected checkpoint and its GEM overall win rate. Checkpoints are selected per variant by the best suite average (the mean of the eight benchmark columns); the corresponding evaluation trajectories appear in Figures 8, 12, and 6.

Table 11: **Full ablation breakdown (games setting, Qwen3-30B-A3B-Instruct-2507).** Best checkpoint per variant on the suite average. AIME reports Avg@32; GPQA-D accuracy; LCB-v6 Pass@1; Reasoning-Gym (RG) win rate at HARD; GEM the overall win rate across the GEM game suite (Liu et al., 2025b). Best in **bold**.

Setting	**Ckpt**	AIME'25	AIME'26	GPQA-D	LCB-v6	RG-Math	RG-Algo.	RG-Cog.	RG-Logic	GEM	**Avg**
Qwen3-30B-A3B-Instruct-2507	–	61.5	73.5	70.4	43.2	45.0	18.0	23.0	67.0	41.0	50.2
SPADE♠	303	**62.8**	74.4	**75.8**	**47.3**	**63.3**	**32.1**	**37.7**	**72.8**	**50.2**	**58.3**
`Environment Designer` w/ learning potential	–	62.4	74.1	74.2	46.1	57.8	27.9	33.3	71.1	47.4	55.9
2-skill curriculum	399	60.6	71.3	71.7	44.7	53.6	26.7	30.5	70.7	46.0	53.7
w/o corpus grounding	111	61.1	74.1	71.8	46.3	51.6	22.3	32.4	68.7	45.1	53.5
w/o memory	111	59.3	**75.0**	72.3	45.7	49.1	22.9	30.7	70.9	42.4	53.2
w/o `Environment Designer` training and memory	271	59.4	73.5	65.8	39.1	22.5	10.0	7.6	46.0	26.4	40.5
Fixed `Environment Designer` (GPT-5.5)	175	59.9	72.8	74.2	42.6	51.2	24.3	30.7	68.0	45.6	53.0

Figure 8 plots the evaluation trajectories behind Table 4: full SPADE stays strongest late in training, while the partial and non-adaptive controls peak early and then fade.

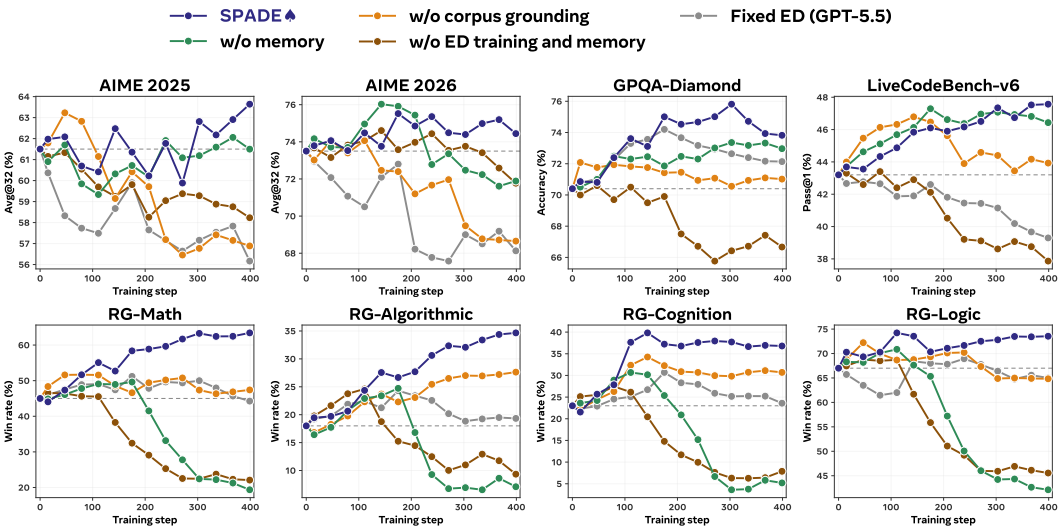


Figure 8: **Removing `Environment Designer` training and memory together drops self-play below base; removing memory or corpus grounding alone yields an above-base selected checkpoint but peaks early and can fall below base late.** One curve per variant of Table 4 (Qwen3-30B-A3B-Instruct-2507, games setting). Top row: AIME 2025/2026 Avg@32, GPQA-Diamond accuracy, and LiveCodeBench-v6 Pass@1; bottom row: the four Reasoning-Gym categories; the dashed line marks the untrained base model.

F.3 Scaling with Model Size

Figure 5 in the main text shows how the gain over each backbone's base grows with model size. Figure 9 tracks the `Environment Designer`'s hint-based regret over training for each backbone size, the signal discussed in Section 4.2.

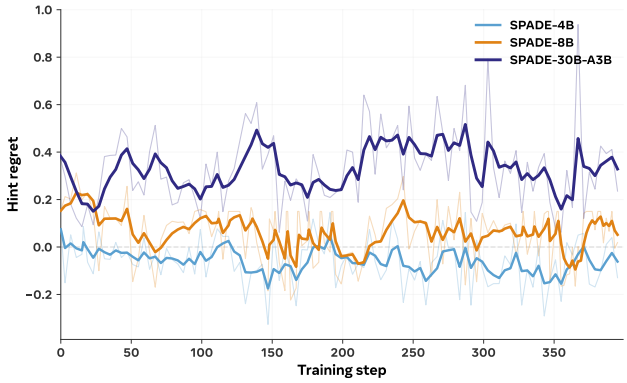


Figure 9: **`Environment Designer` hint-based regret over training across model sizes.** EMA-smoothed (dark) and per-step (light). Only the 30B-A3B estimate stays positive; at 4B it stays below zero for most of training and at 8B it dips below zero intermittently, where the finite-sample estimate turns negative even though regret is non-negative at the optimum (Section 4). Both smaller backbones still gain over base (+5.2, +5.7), so the environments help even where the signal is noisy.

F.4 Curriculum Breadth and a Second Backbone

Curriculum diversity. The cognitive-skill curriculum is the set of skill categories the `Environment Designer` targets when it generates environments: the six listed in Section 5.1, three active per regeneration in round-robin. We compare the full six-skill curriculum against a two-skill version, with everything else held fixed; Figure 10 reports the suite average and Figure 12 the per-benchmark breakdown. The two-skill run also improves, but it captures only about half of the Reasoning-Gym gains and much smaller GPQA-Diamond and LiveCodeBench-v6 gains (best checkpoint on the eight-benchmark suite of Table 1: 53.7 vs. 58.3). The gains grow with the diversity of the curriculum, not with any single game family.

Cross-family transfer. As a check on a non-Qwen backbone, we ran the same recipe on Nemotron-30B-A3B-BF16, evaluating only the Reasoning-Gym suite; AIME 2025/2026, GPQA-Diamond, and LiveCodeBench-v6 were not evaluated on this backbone, so this checks that the training dynamics are not Qwen-specific rather than serving as a matched cross-family comparison. All four Reasoning-Gym categories rise above the untrained base, with the largest Nemotron gains in RG-Cognition (+9.6) and RG-Algorithmic (+9.3; Figure 11).

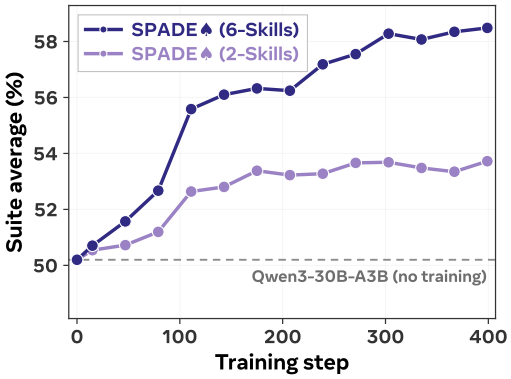


Figure 10: **Curriculum breadth accounts for most of the gain.** Suite average (eight benchmarks of Table 1), six-skill vs. two-skill curriculum. Per-benchmark panels: Figure 12.

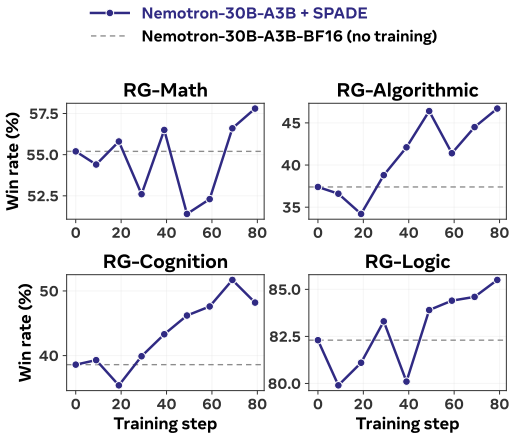


Figure 11: **`SPADE` improves a second backbone family: all four Reasoning-Gym categories end above the untrained Nemotron-30B-A3B-BF16 base (RG-Cognition +9.6, RG-Algorithmic +9.3, RG-Math +2.6, RG-Logic +3.2).** Reasoning-Gym win rate across checkpoints; the dashed line marks the untrained base model. Gains arrive after an initial dip early in training, consistent with the `Environment Designer` initially generating environments too hard for the `Reasoning Agent`, as observed on the Qwen backbones.

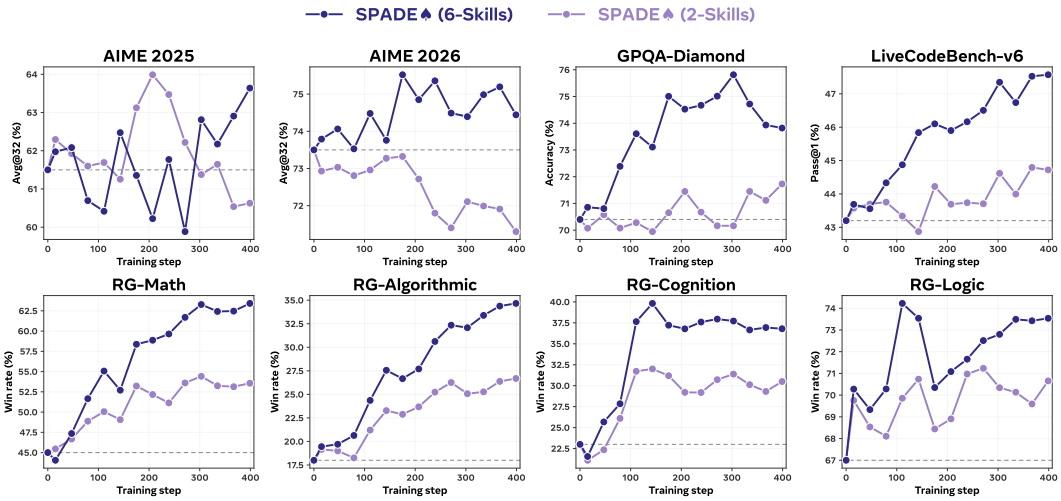


Figure 12: **The full 6-skill curriculum lifts held-out benchmarks more than the restricted 2-skill variant; curriculum breadth drives the gains.** Qwen3-30B-A3B-Instruct-2507, games setting. Top row: AIME 2025/2026 Avg@32, GPQA-Diamond accuracy, and LiveCodeBench-v6 Pass@1. Bottom row: the four Reasoning-Gym categories; the dashed line marks the untrained base model.

F.5 Environment Quality

We track five quality signals; four are tracked over training (Table 12) and executability is measured once over the raw generations. **(i) Learnability (ground truth):** the fraction of environments in the learnable band (`Reasoning Agent` win rate in $[0.2, 0.8]$) rises from 0.16 to 0.31, and `Reasoning Agent` win-rate rises $0.30 \rightarrow 0.62$: the `Environment Designer` keeps targeting tasks at the frontier of the improving agent (Figure 15). **(ii) Well-posedness:** an LLM rubric over all environments scores 97 to 98 % well-posed, flat over training. **(iii) Verifiability:** 90 to 93 % of environments have an objectively checkable terminal answer, flat to slightly rising. **(iv) Executability:** re-executing raw generated code in a sandbox (parse, instantiate, `reset()`), 84.9 % runs as emitted and 90.3 % after stripping a stray Markdown fence; the dominant residual failure is a systematic f-string brace-escaping bug in `\boxed{}` action templates (7.4 %), which the training pipeline's sanitizer repairs before execution (100 % post-filter validity). **(v) Richness:** environments hold steady at ∼320 lines of code, ∼13 hidden state variables, and 8 to 10 interaction turns per episode; the `Environment Designer` does not shortcut to trivial programs to inflate its reward. Reward granularity also rises, from 3.7 to 5.8 distinct levels per environment ($2.2 \rightarrow 4.0$ strictly partial; Figure 13).

Table 12: **Environment quality over training.** Learnability nearly doubles while well-posedness, verifiability, and structural richness hold constant: the quality gains come from sharper difficulty targeting rather than simpler environments. Win-rate rows use the released per-step evaluation logs.

Quality signal	Early (0–40)	Mid (150–250)	Late (340–396)
Learnable-band fraction (win-rate $\in [0.2, 0.8]$)	0.16	0.16	**0.31**
`Reasoning Agent` win-rate	0.30	0.46	**0.62**
Well-posed (LLM rubric)	0.98	0.97	0.97
Verifiable terminal answer (LLM rubric)	0.90	0.91	0.93
Interaction depth (turns / episode)	8.8	8.2	9.8
Program length (lines of code)	316	333	321
Hidden state variables	13.0	13.3	13.4

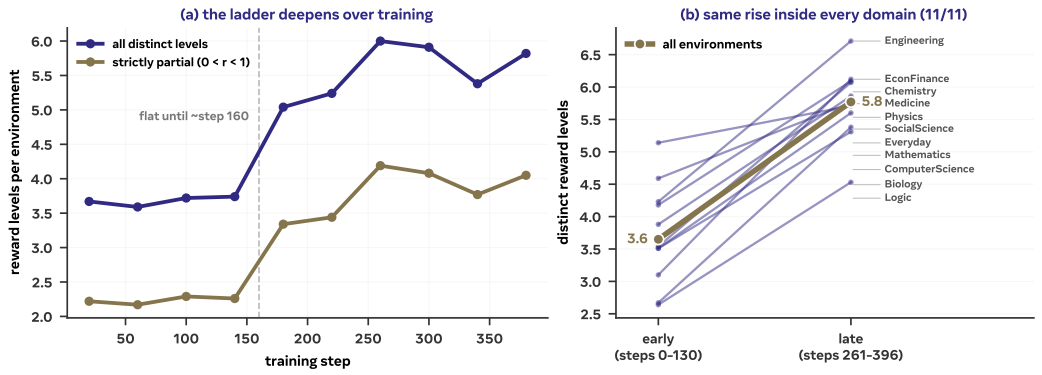


Figure 13: **Reward granularity increases over training.** Left: mean distinct reward levels per environment, including strictly partial levels. Right: early-to-late change in distinct levels overall and by domain (canonical 30B games run).

Figure 14 gives the per-skill decomposition of the `Reasoning Agent`'s gains behind Table 12.

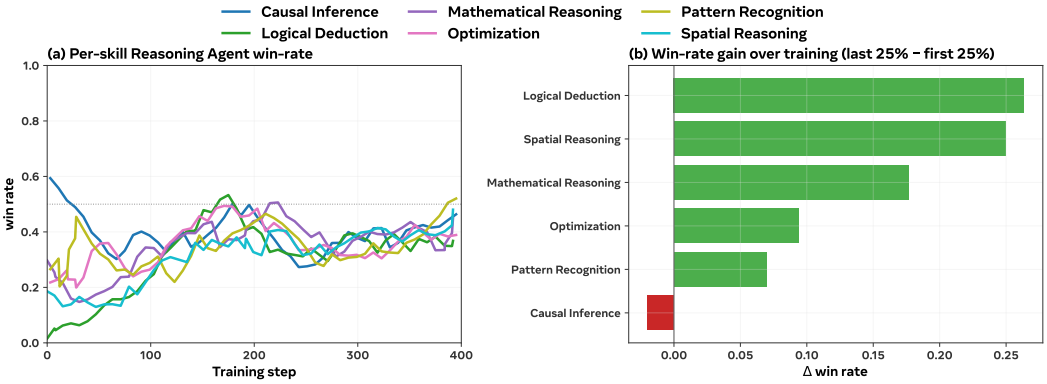


Figure 14: **Per-skill learning** (canonical `SPADE`-30B run). **(a)** Per-skill `Reasoning Agent` win rate over training (EMA over ∼26 logged points/skill). **(b)** Win-rate gain (last 25% minus first 25%): Logical Deduction and Spatial Reasoning improve most; Causal Inference, which starts high, declines slightly.

Figure 15 tracks the learnable share of each rollout over training; full `SPADE` raises it to roughly a third by the end of the run, while the component ablations decline or collapse.

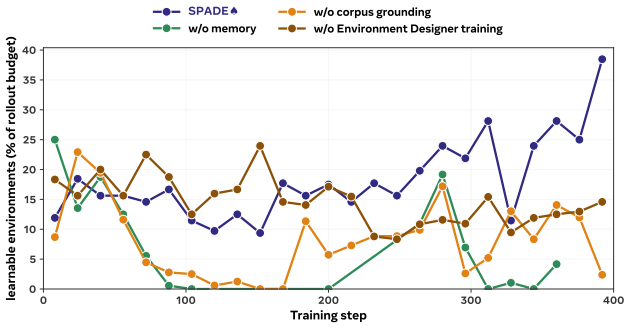


Figure 15: **Full SPADE raises the learnable share of its environment budget to roughly a third by the end of training; component ablations decline or collapse.** Share of each rollout's 24 environments that is *learnable*, defined as `Reasoning Agent` win rate in $[0.2, 0.8]$ and weighted by the number of valid environments generated in each 16-step window. Matched 30B-A3B settings over a common 400-step budget; unfilled rollout capacity contributes zero by construction.

F.6 Environment Diversity

Metric and calibration. Within each training step's batch we report the Vendi Score (Friedman & Dieng, 2023), the effective number of distinct items in a sample,

$$\mathrm{VS}(K) \;=\; \exp\Big(-\textstyle\sum_i \lambda_i \log \lambda_i\Big), \tag{9}$$

where λ_i are the eigenvalues of the normalized cosine-similarity matrix K/n over SBERT embeddings. The score is calibrated: batches of 24 identical environments score 1.0, single-domain batches score 14.8 to 16.5, and mixed batches drawn across the whole run score 21.3 (the practical ceiling for n=24).

Diversity is maintained at the ceiling. Per-step Vendi is flat for the whole run, 20.8 (steps 0 to 40) versus 21.0 (steps 340 to 396), directly at the mixed-population ceiling; mean pairwise cosine distance is likewise flat (0.94). Novelty stays saturated (97 of 100 logged steps consist entirely of never-seen initial states), and all 11 domains of an LLM-labeled taxonomy are present from step 0 with 8.4 per batch of 24 on average (Mathematics 30 %, Physics 20 %, Medicine 11 %, Chemistry 10 %, CS 7 %, Engineering 6 %, other 16 %). The distribution is also stationary: after removing seed-document reuse, a linear probe cannot separate early-half from late-half environments (5-fold AUC 0.551 ± 0.024).

Corpus grounding drives diversity. The ablations quantify the matched-step generation contrast of Appendix G (Table 13, Figure 16). Diversity remains at the mixed-population reference level when memory alone is removed and when `Environment Designer` training and memory are jointly removed, as long as the corpus is retained (0.70 for the joint control); without the corpus, it collapses to 0.04. These configuration-level contrasts associate corpus grounding with preserved diversity, but do not isolate the independent effect of `Environment Designer` training from memory. The windowed dynamics (Figure 16) sharpen this further: the no-corpus run starts near-collapsed (Vendi $\sim$1.6), RL exploration briefly lifts it to a peak of $\sim$5.2 around step 100, and continued optimization then re-collapses it to $\sim$2. That run's best evaluation checkpoint (step 111, Table 11) coincides with its diversity peak; once the environment stream re-collapses, downstream evaluation stops improving. The corpus-grounded runs hold the ceiling for their full run lengths under the same optimization pressure (full `SPADE` for all 400 steps).

Table 13: **Corpus ablations collapse environment diversity.** Vendi Score per 100 environments (SBERT embeddings; mean over 20 balanced draws) on the verified-matched 30B-A3B runs: identical backbone, skill set, and `Environment Designer` system prompt, differing in the listed ablations. The identical analysis under TF-IDF/LSA embeddings reproduces the corpus/no-corpus separation ($0.53/0.58/0.48/0.05$). A fifth run with a static pre-generated environment pool (`Reasoning Agent`-only training, no live `Environment Designer`) is excluded as a different generation protocol; its fixed pool measures Vendi/$n = 0.12$.

Run	Sample	Vendi/$n \uparrow$	Mean pairwise dist. $\uparrow$	Reading
`SPADE` (full)	3,310	0.68	0.94	diverse
w/o memory	3,746	0.69	0.94	diverse
w/o `Environment Designer` training, w/o memory	4,929	**0.70**	0.94	diverse
w/o corpus	866	**0.04**	0.34	collapsed

Scope and limitations. Per-environment scalar rewards are not retained in the public run log, so learnability is measured at the step level (win rate, learnable-band fraction) rather than per environment; the LLM rubric measures absolute difficulty and coherence rather than fit to the current agent; and executability is measured on raw generations, upstream of the pipeline's sanitizer. Diversity numbers use batch size 24 with larger boot-step batches subsampled for comparability.

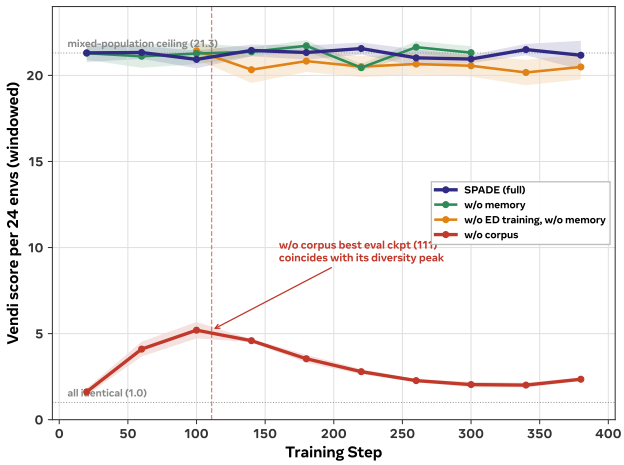


Figure 16: **Environment-diversity dynamics across the verified-matched runs.** Windowed Vendi score (40-step windows, subsampled to exactly 24 environments; mean ± s.d. over 12 draws). The corpus-grounded variants hold the mixed-population reference level for their full run lengths; the no-corpus run starts near-collapsed, peaks at Vendi ∼5.2 near step 100, where its best evaluation checkpoint (111) also falls, and re-collapses as optimization continues. Environment counts differ across runs with run length and acceptance rate; the windowed score uses fixed 24-environment subsamples, so counts do not bias the curves.

G Extended Qualitative Analysis

Summary. Analyzing all 3,310 environments generated by the canonical `SPADE`-30B run, each a distinct program, seeded by one of 1,513 distinct documents drawn from the 15k-document corpus, four results emerge. **(1) Quality:** the share of environments in the learnable band nearly doubles (0.16 → 0.31) while well-posedness, verifiability, and program richness hold constant. **(2) Diversity:** semantic diversity stays at the mixed-population ceiling for all 400 training steps, and the stream keeps producing never-seen environments to the end. **(3) Mechanism:** the diversity comes from corpus grounding; removing the corpus collapses generation to a single task family (Vendi/n: 0.68 vs. 0.04), whereas removing memory alone or jointly removing `Environment Designer` training and memory leaves it intact. **(4) Jointly:** difficulty targeting improves within an unchanged task distribution, so the curriculum sharpens in difficulty while holding its breadth.

We analyze all 3,310 environments generated by the canonical `SPADE`-30B run at the level of their content: the seed document each was generated from, the generated program, and the opening observation. Every generated environment is a distinct program (3,310 of 3,310 unique program hashes; 2,388 distinct initial states; 1,513 distinct seed documents), so all analyses operate on content, never on surface identifiers such as class names (585 distinct; 58 % keep the scaffold default despite the prompt's naming instruction). This appendix presents the qualitative evidence; the matching proxy metrics and training dynamics appear in Appendix F.

G.1 Games

For the games setting we give additional privileged-hint examples and trace how a single environment evolves from early to late training.

G.1.1 Additional Privileged-Hint Examples

Figure 17 shows two positive-regret examples from the canonical 30B games run, each hint beside its task with one logged `Reasoning Agent` rollout with and without it; Figure 18 expands to four task–hint pairs spanning the regret range from large frontier gaps to a small mastery-regime gap.

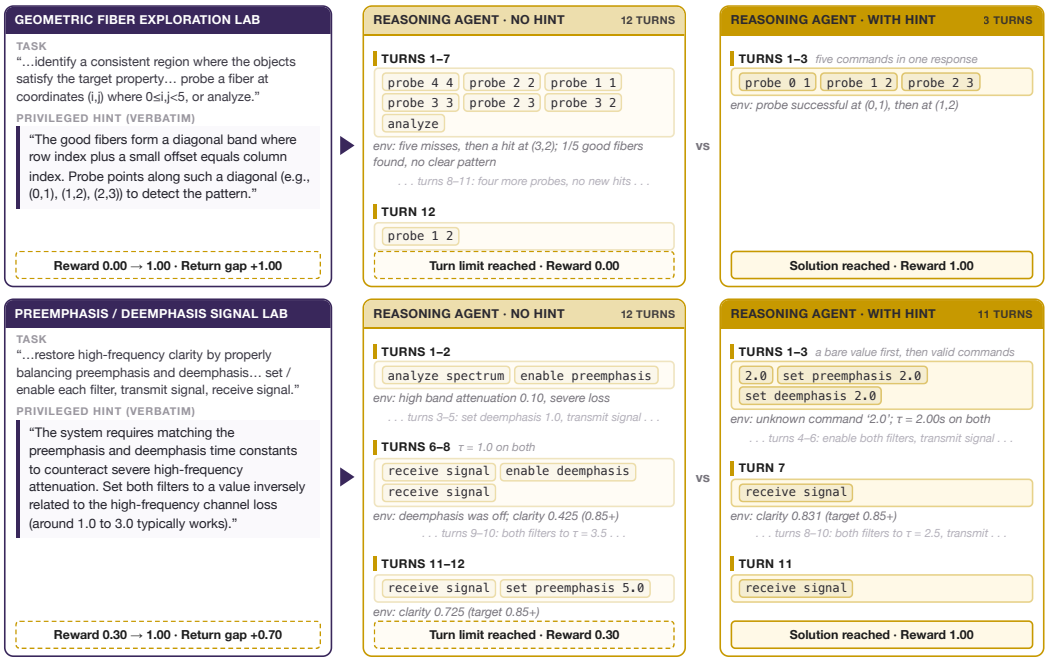


Figure 17: **How a privileged hint changes `Reasoning Agent` play.** Two positive-regret examples from the canonical 30B games run. Left: each environment's task prompt and privileged hint, quoted verbatim (ellipses mark elided text; the standardized answer-format sentence is omitted from the hint). Right: one logged `Reasoning Agent` rollout per arm, condensed while preserving action order, feedback, and values; elided turns are marked and named. The two arms are independent plays with independently seeded resets, so board layouts and probe outcomes differ across arms. The dashed box on each environment card reports the two displayed rollout returns and their single-pair gap; the `Environment Designer` reward in Equation 2 is instead the difference of the arm means over all logged rollouts ($0.00 \rightarrow 1.00$ for the fiber task, $0.30 \rightarrow 0.65$ for the audio task). A larger set of task–hint pairs appears in Figure 18.

Each pair is drawn from one logged game record: the task description is a concise summary of its reset observation, and the hint is a verbatim excerpt of the guidance appended to the with-hint arm.

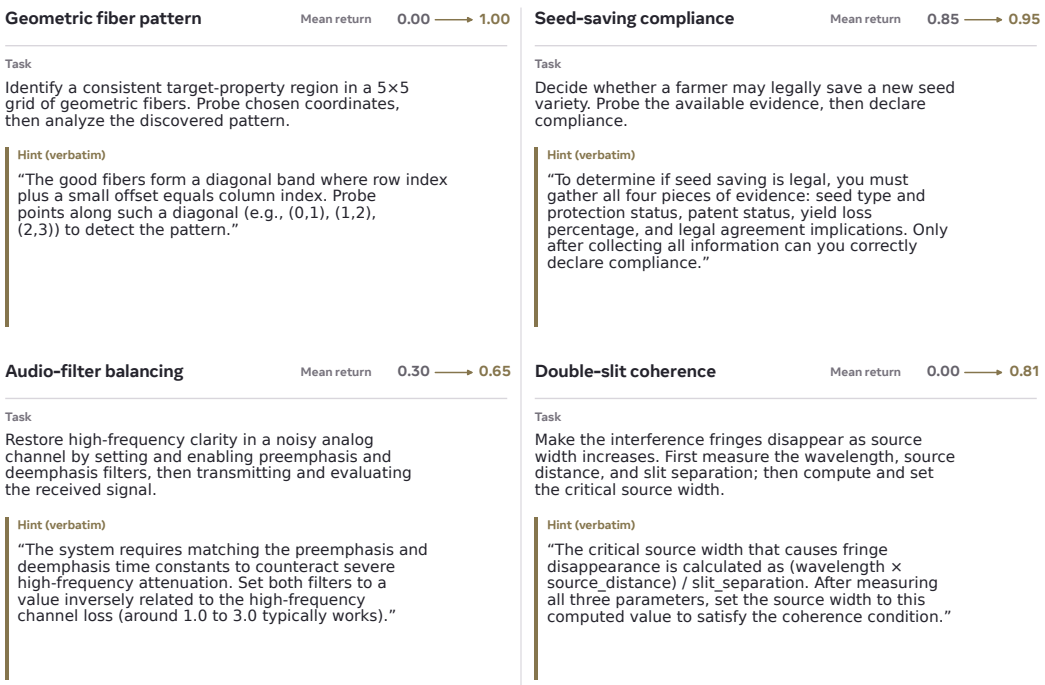


Figure 18: **Task context makes hint utility legible.** Four same-record, positive-regret task–hint pairs from the canonical 30B games run. The task summaries retain the goal, hidden information, and usable interaction while removing generic runtime scaffolding. Hint excerpts omit only the standardized answer-format sentence and runtime wrapper. In each header, the two values report the mean return without hint / with hint.

G.1.2 Environment Evolution over Training

The cards below show one early and one late environment in the exact format produced by the `Environment Designer` (cf. the minimal example in Appendix G.3.1): a seed document grounds the task; the generated program implements a Gym-style interface with hidden state and a verifiable terminal answer; the opening observation states the goal and action space. The corpus's DCLM slice is filtered web text and includes off-topic documents; the early card's personal-finance seed is one such document. The visible shift over training is toward structured, instrumented "laboratory" tasks, while program size, functional hidden state, and verifiability hold steady (about half the early card's raw state variables are write-only narrative fields): the added difficulty comes from state-gated interaction and finer reward grading, not from longer programs.

Training step 0 (early): `CarOwnershipDisputeEnv` — 370 lines, 29 state variables

Seed document (web-scraped): "I have a car, both me and by grandpa are on the title. As I cannot get loan on my own, my grandpa is my cosigner. My grandpa is almost 90, and chances are he may not be around till the loan is paid..."

```python
class CarOwnershipDisputeEnv:
    def __init__(self, max_turns=12, seed=None):
        self.max_turns = max_turns
        self.seed = seed
        self.reset(seed)

    def reset(self, seed=None) -> Tuple[str, dict]:
        self.turn_count = 0
        self.rng = random.Random(seed if seed is not None else 42)

        # Hidden state: the actual legal situation is not fully visible at start
        self.loan_balance = 12000
```

```
        self.car_value = 6000
        self.is_paid_off = False
        self.grandpa_alive = True
        self.grandpa_will_made = self.rng.choice([True, False])  # Hidden: whether
        ↪  will exists
```

Opening observation (what the `Reasoning Agent` sees):

```
You are in a tense family situation.  Your grandfather is on the
title and co-signer of your car loan.  You've made consistent
payments for 3 years, but still owe $12,000 on a car worth only
$6,000.  Your aunt has threatened to take the car when your
grandfather passes.  You're unsure if the will exists or what
it says.  You know your grandfather is still alive and mentally
competent.  You have not yet consulted a lawyer ......
```

Training step 384 (late): `ThermodynamicCycleManipulationLabEnv`
376 lines, 19 state variables

Seed document (web-scraped): "Subject: physics Problem: A gas has N atoms in volume V_0 at temperature T_0. The gas is heated at constant volume up to temperature 3T_0, then allowed to expand isothermally up to volume..."

```
class ThermodynamicCycleManipulationLabEnv:
    def __init__(self, max_turns=12, seed=None):
        self.max_turns = max_turns
        self.seed = seed
        self.reset(seed)

    def reset(self, seed=None) -> Tuple[str, dict]:
        self.turn_count = 0
        self._seed = seed if seed is not None else random.randint(1, 10000)
        random.seed(self._seed)

        # Hidden: thermodynamic system parameters (not directly visible)
        # Represents a monatomic ideal gas undergoing a three-step cycle
        self.n_atoms = random.randint(100, 500)  # N: number of atoms
        self.initial_volume = random.uniform(1.0, 5.0)  # V0: initial volume in m3
        self.initial_temperature = random.uniform(100, 300)  # T0: initial temperature
        ↪  in K
```

Opening observation (what the `Reasoning Agent` sees):

```
You enter a high-precision thermodynamics lab.  Before you is a
transparent cylindrical chamber containing a cloud of gas atoms.
The chamber is sealed and connected to external controls:
Available actions:  - activate heating (starts constant-volume
heating) - initiate expansion (starts isothermal expansion) -
begin cooling (starts constant-pressure cooling) - measure current
entropy (reveals cumulative ......
```

The complete source of both environments (369 and 376 lines as generated) is reproduced verbatim in Appendix G.3.2.

Figures 19 and 20 trace how generation and play change over the run: the `Environment Designer`'s environments stop revealing the solution method in the prompt, and the `Reasoning Agent` shifts from front-loaded derivation toward evidence-first interaction.

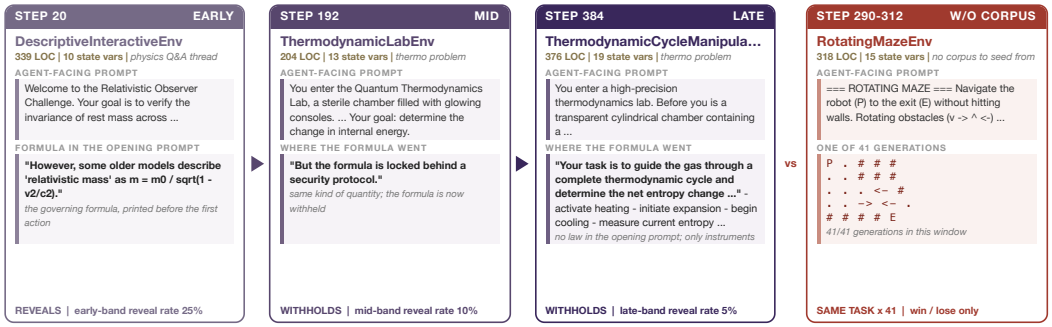


Figure 19: **Trained `Environment Designer` environments stop revealing the solution method in the prompt.** Physics environments from one 30B-A3B run at steps 20, 192, and 384; the step-20 environment retains the scaffold's default class name. The formula-reveal rate (percentage on each panel) falls from 25% to 5% over 473 environments. Rightmost (red): over steps 290–312 the no-corpus ablation emits the same `RotatingMazeEnv` task 41 consecutive times. The complete source of the step-384 environment, together with one further exemplar, appears in Appendix G.3.2.

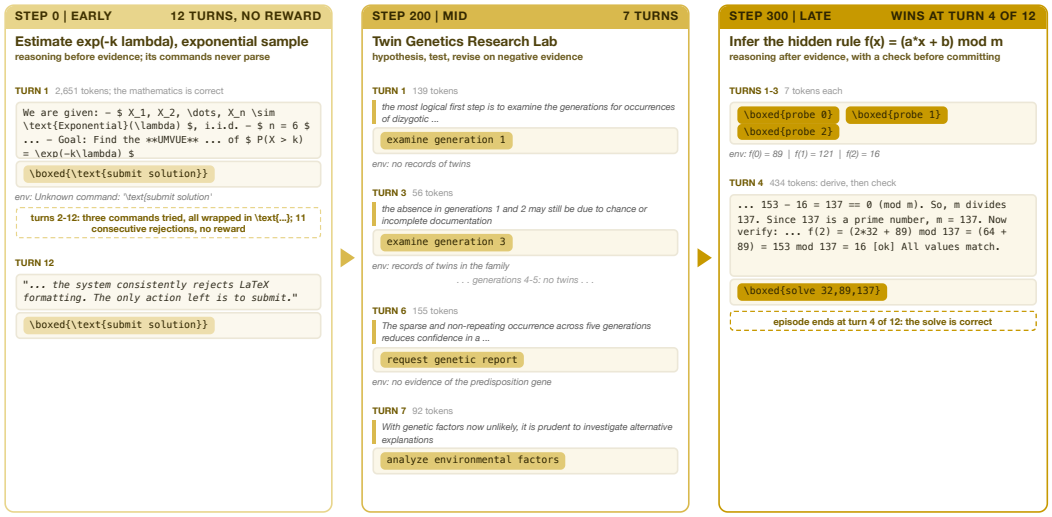


Figure 20: **From front-loaded derivation to evidence-first interaction.** `Reasoning Agent` episodes from one 30B-A3B run at steps 0, 200, and 300. At step 0 the agent derives in advance and cannot recover from format errors; by step 200 it tests short hypotheses and revises on evidence; by step 300 it probes first and derives once. Benchmark gains of late checkpoints (Table 1) confirm the model keeps its long-form derivation ability. Transcripts verbatim (math glyphs transliterated to ASCII; environment feedback abridged); token counts use the backbone's tokenizer.

G.1.3 MATCHED-STEP GENERATION CONTRAST

The clearest qualitative view of diversity is a matched-step contrast against the no-corpus ablation. At training steps 290 to 312, `SPADE` generates environments spanning probability theory, quantum gate tomography, volcanology, hematology, radar signal processing, and hypoelliptic operators, while the no-corpus run generates the *same* rotating-maze navigation task 41 times in a row, varying only the grid layout. Grounding each generation in a sampled corpus document supplies the stream of new task material.

Matched-step generation contrast (training steps 290–312)

`SPADE` (with corpus), one batch at step 296 (24 environments, 24 distinct programs):
`ProbabilitySpaceExplorationEnv`, `QuantumGateSetTomographyLabEnv`, `OptimizationLabEnv`, `NearDoublesMathLabEnv`, `RadarDopplerAnalysisLabEnv`, `MinimaxStrategyLabEnv`, `HypoellipticOperatorLabEnv`, `TrigonometricApproximationLabEnv`, plus 16 default-named environments, each a distinct program on a distinct seed document (pattern recognition, genetics, circuit analysis, ...).

w/o corpus, steps 290–312 (41 environments):
`RotatingMazeEnv`, `RotatingMazeEnv`, `RotatingMazeEnv`, . . . (all 41 generations are the same rotating-obstacle grid-navigation family; only the maze layout and minor rule text vary).

Counts are accepted generations; the two runs' acceptance rates differ over this window.

G.2 TOOL USE

The tool-use `Environment Designer` emits environments in a different register from the games setting: each is a simulated API domain (banking, retail orders, support tickets, telecom accounts, smart-home control) in which a user issues one atomic instruction at a time, and the environment advances only when a programmatic criterion over the hidden state confirms the current instruction is complete. The cards below reproduce one early and one late environment exactly as generated: a code-corpus snippet seeds the generation, the `Environment Designer` builds an unrelated everyday domain on top of it, and each user instruction is paired with a checkable criterion over the backend state (in the early card, coupling the tool call's provenance with its state effect). Criteria check the salient state effect; qualifiers in the instruction text, such as which account pays an invoice, are not always bound, an artifact the validation checks do not catch.

Training step 8 (early): `BankingMultiTurnEnv` **286 lines, 8 tools**

Seed document (web-scraped): "package com.mayo.client.mayoclientapi.persistence.repository; import com.fasterxml.jackson.databind.ObjectMapper; import com.google.api.core.ApiFuture; import com.google.cloud.Timestamp; import com.google.cloud.firestore.;..."*

```
self._user_messages = [
    "List all accounts I have.",
    "Find the checking account and check its current balance.",
    "Transfer $1000 from my savings account to my checking account.",
    "Now pay the unpaid invoice with ID INV2024-003 using my checking account.",
    "Finally, block the card ending in 2468 because it was lost."
]
# Define criteria that must be met for each step to advance
self._message_criteria = [
    lambda s: s['last_searched_account'] is not None and s['last_searched_account'] in
    ↪  s['accounts'],
    lambda s: s['last_query_result'] is not None and
    ↪  s['last_query_result']['account_id'] == 'ACC112233' and
    ↪  s['last_query_result']['balance'] > 0,
    lambda s: s['last_transfer_from'] == 'ACC445566' and s['last_transfer_to'] ==
    ↪  'ACC112233' and s['last_transfer_amount'] == 1000.0,
    lambda s: s['last_invoice_paid'] == 'INV2024-003',
    lambda s: s['last_card_status_change'] == 'blocked'
]
```

Opening observation (what the `Reasoning Agent` sees):

```
List all accounts I have....
```

Training step 392 (late): `CustomerSupportTicketWorkflowEnv` 200 lines, 6 tools

Seed document (web-scraped): "import json import os import sqlite3 import openai from pymongo import MongoClient from pymongo.errors import OperationFailure import tiktoken..."

```
self._user_messages = [
    "Find all high-priority tickets that are still in 'new' status.",
    "Assign the ticket with ID TICKET-001 to agent_01.",
    "Add a note to ticket TICKET-001 stating 'Customer confirmed issue is
    ↪  reproducible.'",
    "Update the status of ticket TICKET-001 to 'in_progress'.",
    "Finally, confirm that ticket TICKET-001 has been successfully resolved by marking
    ↪  it as 'resolved'."
]
self._message_criteria = [
    lambda s: len([t for t in s['tickets'] if t['priority'] == 'high' and t['status']
    ↪  == 'new']) == 2,
    lambda s: any(t['id'] == 'TICKET-001' and t['assigned_to'] == 'agent_01' for t in
    ↪  s['tickets']),
    lambda s: any(t['id'] == 'TICKET-001' and any('Customer confirmed issue is
    ↪  reproducible' in n for n in t['notes']) for t in s['tickets']),
    lambda s: any(t['id'] == 'TICKET-001' and t['status'] == 'in_progress' for t in
    ↪  s['tickets']),
    lambda s: any(t['id'] == 'TICKET-001' and t['status'] == 'resolved' for t in
    ↪  s['tickets'])
]
```

Opening observation (what the `Reasoning Agent` sees):

```
Find all high-priority tickets that are still in 'new' status....
```

Across the full 30B run the structural profile of generated environments is stationary, mirroring the distributional stationarity of the games setting (Appendix F.6): mean program length moves from 247 to 226 lines between the first and last training band, tools per environment from 7.6 to 5.9, while instructions per environment (4.7) and the logical complexity of the per-step criteria (1.9 to 2.0 conditions per criterion) stay flat, and roughly one environment in five ships a guarded failure path (locked cards, unavailable agents, denied requests) that the `Reasoning Agent` must detect and recover from. What changes over training is not the scaffold but the instantiations rotated through it: the `Environment Designer` holds a stable inventory of everyday API domains while regenerating fresh states, identifiers, and instruction sequences each round, exactly the regime the corpus-grounding and memory components are designed to sustain.

G.3 Generated Environment Gallery

This gallery collects representative environments in the exact format the `Environment Designer` emits, starting from a minimal multi-turn example.

G.3.1 A Minimal Generated Environment

Listing 1 shows a minimal multi-turn environment in the format produced by `SPADE`'s `Environment Designer`: a single Python class exposing the Gym-style `reset()`/`step()` interface described in Section 3, with internal state, per-step feedback, and a verifiable terminal reward. This one worked example stands in for the gallery: the same interface and training pipeline also carry single-turn answer-grading tasks and multi-turn tool-use interactions.

Listing 1 A minimal SPADE-generated environment. A Wordle-flavored multi-turn deduction game implemented as a single Python class with Gym-style `reset()`/`step()` interface. The episode is stateful (`self.target`, `self.turns_left`), ends either on a correct guess (terminated, reward 1) or at the 6-turn limit (truncated, reward 0).

```python
import random

class WordleEnv:
    WORDS = ["spade", "trace", "lemon", "graph"]    # truncated

    def reset(self, seed=None):                     # initial state
        self.target = random.Random(seed).choice(self.WORDS)
        self.turns_left = 6
        return "Guess a 5-letter word in 6 tries.", {}

    def step(self, guess):                          # (s', r, term, trunc, info)
        self.turns_left -= 1
        left = [t for g, t in zip(guess, self.target) if g != t]
        fb = ""
        for g, t in zip(guess, self.target):
            if g == t: fb += "G"
            elif g in left: fb += "Y"; left.remove(g)
            else: fb += "-"
        if fb == "GGGGG":
            return fb, 1.0, True, False, {}
        return fb, 0.0, False, self.turns_left == 0, {}
```

G.3.2 Full Source of Two Exemplar Generated Environments

The two environments carded in Appendix G.1.2 are reproduced here in full, exactly as emitted by the `Environment Designer` (lightly transliterated to ASCII: typographic quotes, dashes, arrows, and sub-/superscripts). Each is a complete, self-contained program: hidden state initialized in `reset()`, a multi-turn `step()` loop with intermediate feedback, and a verifiable terminal reward. Episode return is the final-step reward, clipped to $[-1, 1]$ and 0 if the episode never terminates; the intermediate per-step rewards visible below are progress feedback, not summed.

Training step 0: `CarOwnershipDisputeEnv` (370 lines).

```python
import random
import re
from typing import Tuple, Dict

class CarOwnershipDisputeEnv:
    def __init__(self, max_turns=12, seed=None):
        self.max_turns = max_turns
        self.seed = seed
        self.reset(seed)

    def reset(self, seed=None) -> Tuple[str, dict]:
        self.turn_count = 0
        self.rng = random.Random(seed if seed is not None else 42)

        # Hidden state: the actual legal situation is not fully visible at start
        self.loan_balance = 12000
        self.car_value = 6000
        self.is_paid_off = False
        self.grandpa_alive = True
        self.grandpa_will_made = self.rng.choice([True, False])  # Hidden: whether will exists
        self.grandpa_will_content = self.rng.choice(['gives car to OP', 'gives car to aunt', 'no mention']) if self.grandpa_will_made else None
        self.grandpa_owns_title = True  # Co-owner
        self.op_owns_title = True
        self.loan_coborrower = True
        self.loan_cosigner = True
        self.loan_lender_has_lien = True
        self.title_in_name = True  # Title exists in both names
        self.loan_docs_contain_death_clause = self.rng.choice([True, False])
        self.aunt_threatening = self.rng.choice([True, False])
        self.op_paying_payments = True
        self.op_has_good_payment_history = True
        self.op_credit_score = 580  # Poor
```

```
        self.op_has_refinancing_attempt = False
        self.op_has_consulted_lawyer = False
        self.grandpa_is_competent = True
        self.grandpa_will_likely_sign = self.rng.choice([True, False])
        self.grandpa_has_signed_transfer = False
        self.grandpa_has_refinanced = False
        self.grandpa_has_gifted_title = False
        self.legal_right_to_car = True  # OP has right due to ownership and payments

        # Goal: Ensure OP keeps the car after Grandpa's death
        # This requires multiple steps: verify legal rights, assess will, consider refinancing, or get title transfer.

        # Initial observation - hidden state, partial information
        obs = (
            "You are in a tense family situation. Your grandfather is on the title and co-signer of your car loan. "
            "You've made consistent payments for 3 years, but still owe $12,000 on a car worth only $6,000. "
            "Your aunt has threatened to take the car when your grandfather passes. "
            "You're unsure if the will exists or what it says. "
            "You know your grandfather is still alive and mentally competent. "
            "You have not yet consulted a lawyer or attempted refinancing. "
            "The loan documents may have a clause about death of a co-signer.\n\n"
            "Available actions:\n"
            "  - check loan documents\n"
            "  - talk to grandpa about the will\n"
            "  - talk to grandpa about transferring title\n"
            "  - research refinancing options\n"
            "  - consult a lawyer\n"
            "  - check car title status\n"
            "  - review state laws on joint ownership\n\n"
            "Reply with your next action as \\boxed{<action>}."
        )
        return obs, {}

    def step(self, action: str) -> Tuple[str, float, bool, bool, dict]:
        self.turn_count += 1
        truncated = self.turn_count >= self.max_turns
        m = re.search(r'\\boxed\{(.+?)\}', action)
        cmd = (m.group(1) if m else action).strip().lower()

        reward = 0.0
        terminated = False
        info = {}

        # Parse and execute command
        if cmd == "check loan documents":
            if self.loan_docs_contain_death_clause:
                obs = (
                    "You reviewed the loan documents. There is a clause: 'If either borrower dies, the remaining balance becomes due immediately.'\n"
                    "This means if your grandfather dies, you may have to pay off the full $12,000 immediately unless refinanced.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - research refinancing options\n"
                    "  - consult a lawyer\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.2
            else:
                obs = (
                    "You reviewed the loan documents. There is no death clause. The loan remains active and payable by you regardless of your grandfather's passing.\n"
                    "This is good news - you are not immediately liable for the full balance.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about the will\n"
                    "  - check car title status\n"
                    "  - talk to grandpa about transferring title\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.2

        elif cmd == "talk to grandpa about the will":
            if not self.grandpa_alive:
                obs = (
                    "Your grandfather has passed. You cannot talk to him now.\n"
                    "You must now rely on the will or state law.\n\n"
                    "Next steps:\n"
                    "  - check if a will exists\n"
```

```
                    "  - consult a lawyer\n"
                    "  - verify car title status\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.1
            elif self.grandpa_will_made:
                obs = (
                    "You talked to your grandfather. He confirms he has a will. He says he
                    ↪  intends to leave the car to you.\n"
                    "He will sign the will soon.\n\n"
                    "Next steps:\n"
                    "  - verify the will's content\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - consult a lawyer\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.3
            else:
                obs = (
                    "You talked to your grandfather. He says he hasn't made a will yet. He
                    ↪  is open to writing one.\n"
                    "He is willing to transfer the car to you before he dies.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - consult a lawyer\n"
                    "  - research title transfer process\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.2

        elif cmd == "talk to grandpa about transferring title":
            if not self.grandpa_alive:
                obs = (
                    "Your grandfather has passed. You cannot talk to him now.\n"
                    "You must now rely on the will or state law.\n\n"
                    "Next steps:\n"
                    "  - check if a will exists\n"
                    "  - consult a lawyer\n"
                    "  - verify car title status\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.1
            elif self.grandpa_will_made and self.grandpa_will_content == "gives car to
            ↪  OP":
                obs = (
                    "You talked to your grandfather. He confirms he has a will leaving the
                    ↪  car to you.\n"
                    "He agrees to transfer the title to you now.\n"
                    "You can now begin the process to remove his name from the title.\n\n"
                    "Next steps:\n"
                    "  - check car title status\n"
                    "  - research title transfer process\n"
                    "  - consult a lawyer\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.4
                self.grandpa_has_gifted_title = True
            elif self.grandpa_will_made and self.grandpa_will_content == "gives car to
            ↪  aunt":
                obs = (
                    "You talked to your grandfather. He confirms he has a will leaving the
                    ↪  car to your aunt.\n"
                    "He says he cannot change it now, but he is willing to transfer the
                    ↪  title to you as a gift.\n"
                    "This may be legally possible, but could trigger tax or probate
                    ↪  issues.\n\n"
                    "Next steps:\n"
                    "  - consult a lawyer\n"
                    "  - research gift transfer options\n"
                    "  - check if lender allows title change\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.2
                self.grandpa_has_gifted_title = True
            elif not self.grandpa_will_made and self.grandpa_has_refinanced:
                obs = (
                    "You talked to your grandfather. He has already refinanced the loan in
                    ↪  his name only.\n"
                    "The loan is now in his name, and the car is titled solely in your
                    ↪  name.\n"
                    "This secures the car for you.\n\n"
                    "Next steps:\n"
                    "  - verify title transfer with DMV\n"
```

```
                    "  - confirm lender has released lien\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.5
            elif not self.grandpa_will_made and self.grandpa_will_likely_sign:
                obs = (
                    "You talked to your grandfather. He agrees to transfer the title to
                    ↪  you.\n"
                    "He will sign the necessary documents soon.\n"
                    "This is the most secure way to ensure you keep the car.\n\n"
                    "Next steps:\n"
                    "  - check car title status\n"
                    "  - consult a lawyer\n"
                    "  - begin title transfer process\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.3
                self.grandpa_has_gifted_title = True
            else:
                obs = (
                    "You talked to your grandfather. He agrees to transfer the title to
                    ↪  you.\n"
                    "He is willing to sign the documents.\n"
                    "You can now proceed with the title transfer.\n\n"
                    "Next steps:\n"
                    "  - check car title status\n"
                    "  - consult a lawyer\n"
                    "  - begin title transfer process\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.3
                self.grandpa_has_gifted_title = True

        elif cmd == "research refinancing options":
            if self.op_credit_score >= 650 and self.op_paying_payments and
            ↪  self.op_has_good_payment_history:
                obs = (
                    "You researched refinancing. Your credit is still poor, but your
                    ↪  payment history is strong.\n"
                    "Some credit unions may consider you for refinancing.\n"
                    "You should apply to a local credit union.\n\n"
                    "Next steps:\n"
                    "  - apply to credit union for refinancing\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - consult a lawyer\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.2
            else:
                obs = (
                    "You researched refinancing. Your credit score is still too low to
                    ↪  qualify.\n"
                    "The loan is underwater, making refinancing difficult.\n"
                    "You may need to wait until your credit improves or your grandfather
                    ↪  transfers the title.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - consult a lawyer\n"
                    "  - check car title status\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.1

        elif cmd == "consult a lawyer":
            if not self.op_has_consulted_lawyer:
                obs = (
                    "You consulted a lawyer. They confirmed:\n"
                    "- You have a legal right to the car as long as you keep paying.\n"
                    "- The title is in both names, so you can't sell it without his
                    ↪  consent.\n"
                    "- A will can override joint ownership, but it's not automatic.\n"
                    "- You should get the title transferred to you now.\n"
                    "- A gift or transfer is possible while he's alive.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - check car title status\n"
                    "  - begin transfer process\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.4
                self.op_has_consulted_lawyer = True
            else:
                obs = (
```

```
                "You've already consulted a lawyer. They advised that you should transfer the title to your name now.\n"
                "This is the best way to secure the car.\n\n"
                "Next steps:\n"
                "  - talk to grandpa about transferring title\n"
                "  - check car title status\n"
                "  - begin transfer process\n"
                "Reply with \\boxed{<action>}."
            )
            reward = 0.1

        elif cmd == "check car title status":
            if self.grandpa_has_gifted_title:
                obs = (
                    "You checked the title status. The title has been transferred to you. Your grandfather signed it.\n"
                    "The car is now in your name only.\n"
                    "You are safe from the aunt's threat.\n\n"
                    "You have secured the car.\n"
                    "Goal achieved! Reward: 1.0"
                )
                reward = 1.0
                terminated = True
            elif self.grandpa_has_refinanced:
                obs = (
                    "You checked the title status. The loan is now in your grandfather's name only.\n"
                    "The title is still in both names, but the lender has released the lien.\n"
                    "You must now transfer the title to your name.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - consult a lawyer\n"
                    "  - begin title transfer process\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.3
            else:
                obs = (
                    "You checked the title status. The title is still in both your names.\n"
                    "The lender holds a lien until the loan is paid off.\n"
                    "You cannot sell or fully transfer the car without their consent.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - consult a lawyer\n"
                    "  - research title transfer process\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.1

        elif cmd == "review state laws on joint ownership":
            if random.Random(str(self.grandpa_will_content)).random() < 0.5:  # State law fixed per episode
                obs = (
                    "You reviewed state laws. In your state, joint ownership means the surviving co-owner automatically inherits the car.\n"
                    "Even if your grandfather's will says otherwise, the car goes to you.\n"
                    "This is a major advantage.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about transferring title\n"
                    "  - consult a lawyer\n"
                    "  - check car title status\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.3
            else:
                obs = (
                    "You reviewed state laws. In your state, joint ownership does not automatically transfer the car.\n"
                    "The car becomes part of the estate and is subject to the will.\n"
                    "You must ensure your grandfather leaves the car to you in his will.\n\n"
                    "Next steps:\n"
                    "  - talk to grandpa about the will\n"
                    "  - consult a lawyer\n"
                    "  - research will options\n"
                    "Reply with \\boxed{<action>}."
                )
                reward = 0.2

        else:
```

```
            # Unrecognized command
            obs = (
                "Invalid action. Please choose from:\n"
                "  - check loan documents\n"
                "  - talk to grandpa about the will\n"
                "  - talk to grandpa about transferring title\n"
                "  - research refinancing options\n"
                "  - consult a lawyer\n"
                "  - check car title status\n"
                 "  - review state laws on joint ownership\n\n"
                "Reply with \\boxed{<action>}."
            )
            reward = 0.0

        # Check if goal is reached: car is in OP's name only
        if self.grandpa_has_gifted_title or (self.grandpa_has_refinanced and
        ↪  self.op_owns_title and not self.grandpa_owns_title):
            obs = (
                "Congratulations! You have successfully secured the car.\n"
                "The title has been transferred to you, and your grandfather is no longer
                ↪  on it.\n"
                "The threat from your aunt is now irrelevant.\n"
                "You are in full control of the vehicle.\n"
                "Goal achieved! Reward: 1.0"
            )
            reward = 1.0
            terminated = True

        return obs, reward, terminated, truncated, info

    def solution(self) -> str:
        return (
            "1. Talk to grandpa about the will.\n"
            "2. If will exists and leaves car to OP, proceed to transfer.\n"
            "3. If no will, talk to grandpa about transferring title.\n"
            "4. Consult a lawyer to confirm legal rights.\n"
            "5. Check car title status.\n"
            "6. Transfer title via gift or refinancing.\n"
            "7. Confirm lender has released lien.\n"
            "8. Ensure car is titled solely in OP's name."
        )

    def close(self):
        pass
```

Training step 384: `ThermodynamicCycleManipulationLabEnv` (376 lines).

```
"""Interactive Thermodynamic Cycle Manipulation Lab: A Multi-Turn Mathematical Reasoning
↪  Environment"""

import random
import re
import math
from typing import Tuple, Dict

class ThermodynamicCycleManipulationLabEnv:
    def __init__(self, max_turns=12, seed=None):
        self.max_turns = max_turns
        self.seed = seed
        self.reset(seed)

    def reset(self, seed=None) -> Tuple[str, dict]:
        self.turn_count = 0
        self._seed = seed if seed is not None else random.randint(1, 10000)
        random.seed(self._seed)

        # Hidden: thermodynamic system parameters (not directly visible)
        # Represents a monatomic ideal gas undergoing a three-step cycle
        self.n_atoms = random.randint(100, 500)  # N: number of atoms
        self.initial_volume = random.uniform(1.0, 5.0)  # V0: initial volume in m3
        self.initial_temperature = random.uniform(100, 300)  # T0: initial temperature in
        ↪  K
        self.kb = 1.380649e-23  # Boltzmann constant (J/K), fixed physical constant

        # Internal state of the lab equipment and system conditions
        self.current_step = "idle"  # idle, heating, expansion, cooling, complete
        self.current_volume = self.initial_volume
        self.current_temperature = self.initial_temperature
        self.pressure = (self.n_atoms * self.kb * self.current_temperature) /
        ↪  self.current_volume
```

```python
        self.entropy_current = 0.0  # cumulative entropy change so far (in J/K)
        self.entropy_history = []  # stores entropy change per step for tracking
        self.heating_complete = False
        self.expansion_complete = False
        self.cooling_complete = False
        self.target_entropy_change = 0.0  # will be computed as part of the cycle

        # Compute the expected net entropy change (0) via the physics of the cycle
        # This is hidden - agent must discover it through manipulation
        # DeltaS_net = (3/2)NkB*ln(3) + NkB*ln(3) - (5/2)NkB*ln(3) = 0
        self.target_entropy_change = 0.0

        # Goal: manipulate the system through three thermodynamic processes
        # and observe that the net entropy change is zero (cyclic process)
        self.goal_achieved = False

        # Initial observation: player enters a climate-controlled lab with sealed gas chamber
        observation = (
            "You enter a high-precision thermodynamics lab. Before you is a transparent cylindrical chamber "
            "containing a cloud of gas atoms. The chamber is sealed and connected to external controls:\n\n"
            "Available actions:\n"
            "  - activate heating (starts constant-volume heating)\n"
            "  - initiate expansion (starts isothermal expansion)\n"
            "  - begin cooling (starts constant-pressure cooling)\n"
            "  - measure current entropy (reveals cumulative entropy change so far)\n"
            "  - scan system parameters (reveals current volume, temperature, pressure)\n\n"
            "The gas is initially at equilibrium. Your task is to guide the gas through a complete thermodynamic cycle "
            "and determine the net entropy change by observing the system's behavior across multiple steps."
        )
        return observation, {}

    def step(self, action: str) -> Tuple[str, float, bool, bool, dict]:
        self.turn_count += 1
        truncated = self.turn_count >= self.max_turns
        m = re.search(r'\\boxed\{(.+?)\}', action)
        cmd = (m.group(1) if m else action).strip().lower()

        # Default reward and termination
        reward = 0.0
        terminated = False
        info = {}

        # Parse and execute command
        if self.current_step == "idle":
            if "activate heating" in cmd:
                # Step 1: Heat at constant volume from T0 to 3T0
                self.current_temperature = 3.0 * self.initial_temperature
                # Volume remains constant
                self.current_volume = self.initial_volume
                # Pressure increases proportionally
                self.pressure = (self.n_atoms * self.kb * self.current_temperature) / self.current_volume
                # Calculate entropy change for constant-volume heating: DeltaS = (3/2)NkB*ln(3)
                delta_s1 = 1.5 * self.n_atoms * self.kb * math.log(3.0)
                self.entropy_current += delta_s1
                self.entropy_history.append(delta_s1)
                self.heating_complete = True
                self.current_step = "heating"
                observation = (
                    f"You activate the heating system. The gas is now being heated at constant volume.\n\n"
                    f"Current state:\n"
                    f"  Volume: {self.current_volume:.3f} m3 (constant)\n"
                    f"  Temperature: {self.current_temperature:.1f} K (increased to 3x initial)\n"
                    f"  Pressure: {self.pressure:.2e} Pa\n"
                    f"  Entropy change (step 1): +{delta_s1:.4e} J/K\n\n"
                    f"Next available actions:\n"
                    f"  - initiate expansion (to start isothermal expansion)\n"
                    f"  - measure current entropy (to check cumulative change)\n"
                    f"  - scan system parameters (view detailed state)"
                )
                reward = 0.3  # partial progress: first step complete

            elif "measure current entropy" in cmd:
                observation = (
```

```
                    f"You measure the current entropy using the quantum calorimeter.\n\n"
                    f"Total entropy change so far: {self.entropy_current:.4e} J/K\n\n"
                    f"Note: The system is still in initial state. No processes have been initiated yet.\n\n"
                    f"Available actions:\n"
                    f"  - activate heating (to begin the first process)\n"
                    f"  - scan system parameters (to inspect hidden variables)"
                )
                reward = 0.1  # small progress for probing

            elif "scan system parameters" in cmd:
                # Reveal hidden internal values (but not the full solution)
                observation = (
                    f"You run a diagnostic scan on the system.\n\n"
                    f"Internal parameters (hidden):\n"
                    f"  Number of atoms: {self.n_atoms}\n"
                    f"  Initial volume: {self.initial_volume:.3f} m3\n"
                    f"  Initial temperature: {self.initial_temperature:.1f} K\n"
                    f"  Boltzmann constant: {self.kb:.6e} J/K\n\n"
                    f"Current operational state: idle (waiting for heating command)\n\n"
                    f"Available actions:\n"
                    f"  - activate heating (to start the cycle)\n"
                    f"  - measure current entropy (to track changes)"
                )
                reward = 0.2  # moderate progress for exploration

            else:
                observation = (
                    f"Invalid command: '{cmd}'.\n\n"
                    f"Available actions in idle state:\n"
                    f"  - activate heating\n"
                    f"  - measure current entropy\n"
                    f"  - scan system parameters"
                )
                reward = 0.0

        elif self.current_step == "heating" and self.heating_complete:
            if "initiate expansion" in cmd:
                # Step 2: Isothermal expansion from V0 to 3V0 at 3T0
                self.current_volume = 3.0 * self.initial_volume
                # Temperature remains constant at 3T0
                self.current_temperature = 3.0 * self.initial_temperature
                # Pressure decreases inversely with volume
                self.pressure = (self.n_atoms * self.kb * self.current_temperature) / self.current_volume
                # Calculate entropy change for isothermal expansion: DeltaS = NkB*ln(3)
                delta_s2 = self.n_atoms * self.kb * math.log(3.0)
                self.entropy_current += delta_s2
                self.entropy_history.append(delta_s2)
                self.expansion_complete = True
                self.current_step = "expansion"
                observation = (
                    f"You initiate the isothermal expansion. The chamber expands while maintaining temperature.\n\n"
                    f"Current state:\n"
                    f"  Volume: {self.current_volume:.3f} m3 (tripled)\n"
                    f"  Temperature: {self.current_temperature:.1f} K (constant)\n"
                    f"  Pressure: {self.pressure:.2e} Pa (reduced)\n"
                    f"  Entropy change (step 2): +{delta_s2:.4e} J/K\n\n"
                    f"Next available actions:\n"
                    f"  - begin cooling (to start constant-pressure cooling)\n"
                    f"  - measure current entropy (to verify cumulative change)\n"
                    f"  - scan system parameters (view updated state)"
                )
                reward = 0.6  # progress toward goal

            elif "measure current entropy" in cmd:
                observation = (
                    f"You measure the cumulative entropy change.\n\n"
                    f"Total entropy change so far: {self.entropy_current:.4e} J/K\n\n"
                    f"System is ready for expansion. The gas has been heated; expansion has not yet begun.\n\n"
                    f"Available actions:\n"
                    f"  - initiate expansion (next step)\n"
                    f"  - begin cooling (to continue the cycle)\n"
                    f"  - scan system parameters"
                )
                reward = 0.4

            elif "scan system parameters" in cmd:
                observation = (
                    f"Diagnostic scan during heating phase:\n\n"
                    f"Current state:\n"
```

```
                    f"  Volume: {self.current_volume:.3f} m3\n"
                    f"  Temperature: {self.current_temperature:.1f} K\n"
                    f"  Pressure: {self.pressure:.2e} Pa\n"
                    f"  Atoms: {self.n_atoms}\n"
                    f"  Initial conditions: V0={self.initial_volume:.3f},
                    ↪  T0={self.initial_temperature:.1f}\n\n"
                    f"Entropy changes recorded:\n"
                    f"  Step 1 (heating): +{self.entropy_history[0]:.4e} J/K\n"
                    f"  Steps recorded so far: {len(self.entropy_history)}\n\n"
                    f"Next step: cooling at constant pressure."
                )
                reward = 0.5

            else:
                observation = (
                    f"Invalid command: '{cmd}'.\n\n"
                    f"Available actions during expansion:\n"
                    f"  - initiate expansion (already active)\n"
                    f"  - begin cooling\n"
                    f"  - measure current entropy\n"
                    f"  - scan system parameters"
                )
                reward = 0.0

        elif self.current_step == "expansion" and self.expansion_complete:
            if "begin cooling" in cmd:
                # Step 3: Cool at constant pressure from 3T0 to T0
                # Pressure is held constant at the expanded state's pressure
                target_temp = self.initial_temperature  # T0
                self.current_temperature = target_temp
                # Volume must change to maintain constant pressure: V ~ T
                self.current_volume = self.current_volume * (target_temp / (3.0 *
                ↪  self.initial_temperature))
                # Pressure remains constant
                self.pressure = (self.n_atoms * self.kb * self.current_temperature) /
                ↪  self.current_volume
                # Calculate entropy change for constant-pressure cooling: DeltaS =
                ↪  (5/2)NkB*ln(1/3)
                delta_s3 = 2.5 * self.n_atoms * self.kb * math.log(1.0 / 3.0)  # negative
                ↪  value
                self.entropy_current += delta_s3
                self.entropy_history.append(delta_s3)
                self.cooling_complete = True
                self.current_step = "cooling"
                observation = (
                    f"You begin the cooling phase at constant pressure.\n\n"
                    f"Current state:\n"
                    f"  Volume: {self.current_volume:.3f} m3 (reduced)\n"
                    f"  Temperature: {self.current_temperature:.1f} K (returned to
                    ↪  initial)\n"
                    f"  Pressure: {self.pressure:.2e} Pa (constant)\n"
                    f"  Entropy change (step 3): {delta_s3:.4e} J/K (decrease)\n\n"
                    f"Cycle complete. The system has returned to its original
                    ↪  temperature.\n\n"
                    f"Final available actions:\n"
                    f"  - measure current entropy (to determine net change)\n"
                    f"  - scan system parameters (verify full cycle)"
                )
                reward = 0.9  # almost complete

            elif "measure current entropy" in cmd:
                observation = (
                    f"You measure the total entropy change after the expansion.\n\n"
                    f"Total entropy change: {self.entropy_current:.4e} J/K\n\n"
                    f"Note: The cooling step has not yet been performed.\n\n"
                    f"Available actions:\n"
                    f"  - begin cooling (to finalize)\n"
                    f"  - scan system parameters"
                )
                reward = 0.7

            elif "scan system parameters" in cmd:
                observation = (
                    f"Diagnostic scan before cooling:\n\n"
                    f"Final state:\n"
                    f"  Volume: {self.current_volume:.3f} m3\n"
                    f"  Temperature: {self.current_temperature:.1f} K\n"
                    f"  Pressure: {self.pressure:.2e} Pa\n"
                    f"  Atoms: {self.n_atoms}\n\n"
                    f"Entropy changes:\n"
                    f"  Heating: +{self.entropy_history[0]:.4e}\n"
                    f"  Expansion: +{self.entropy_history[1]:.4e}\n"
                    f"  Cooling: pending\n\n"
```

```
                    f"Net entropy change so far: {self.entropy_current:.4e} J/K"
                )
                reward = 0.8

            else:
                observation = (
                    f"Invalid command: '{cmd}'.\n\n"
                    f"Available actions during cooling:\n"
                    f"  - begin cooling (already started)\n"
                    f"  - measure current entropy\n"
                    f"  - scan system parameters"
                )
                reward = 0.0

        elif self.current_step == "cooling" and self.cooling_complete:
            if "measure current entropy" in cmd:
                # Final check: assess net entropy change
                net_entropy = self.entropy_current
                # Due to floating point precision, we accept small deviation
                if abs(net_entropy) < 1e-20:
                    self.goal_achieved = True
                    terminated = True
                    reward = 1.0
                    observation = (
                        f"YOU HAVE SUCCESSFULLY COMPLETED THE THERMODYNAMIC CYCLE.\n\n"
                        f"Final entropy measurement:\n"
                        f"  Net entropy change: {net_entropy:.2e} J/K\n\n"
                        f"This confirms the system has returned to its original state with
                        ↪  no net entropy change,\n"
                        f"as expected for a reversible cyclic process in an ideal
                        ↪  gas.\n\n"
                        f"Congratulations! You've demonstrated the mathematical reasoning
                        ↪  behind entropy conservation "
                        f"in cyclic thermodynamic processes.\n\n"
                        f"Your solution is valid and complete. This episode is
                        ↪  terminated."
                    )
                else:
                    # Still not zero - agent needs to persist
                    observation = (
                        f"You measure the final entropy change.\n\n"
                        f"Net entropy change: {net_entropy:.4e} J/K (not zero)\n\n"
                        f"The system has completed all three processes, but the net
                        ↪  entropy is not yet balanced.\n"
                        f"Check each recorded step against the formula for its
                        ↪  process.\n\n"
                        f"Try verifying your steps or re-running measurements with higher
                        ↪  precision."
                    )
                    reward = 0.95  # very high partial reward, but not complete

            elif "scan system parameters" in cmd:
                observation = (
                    f"Final system status scan:\n\n"
                    f"Cycle complete. All processes executed:\n"
                    f"  - Constant-volume heating: T -> 3T0\n"
                    f"  - Isothermal expansion: V -> 3V0\n"
                    f"  - Constant-pressure cooling: T -> T0\n\n"
                    f"Final state variables:\n"
                    f"  Volume: {self.current_volume:.3f} m3\n"
                    f"  Temperature: {self.current_temperature:.1f} K\n"
                    f"  Pressure: {self.pressure:.2e} Pa\n\n"
                    f"Entropy changes:\n"
                    f"  Step 1: +{self.entropy_history[0]:.4e}\n"
                    f"  Step 2: +{self.entropy_history[1]:.4e}\n"
                    f"  Step 3: {self.entropy_history[2]:.4e}\n"
                    f"  Total: {self.entropy_current:.4e} J/K\n\n"
                    f"Verify your result with a final entropy measurement."
                )
                reward = 0.9

            else:
                observation = (
                    f"Invalid command: '{cmd}'.\n\n"
                    f"Final actions available:\n"
                    f"  - measure current entropy (to check net change)\n"
                    f"  - scan system parameters (for detailed verification)"
                )
                reward = 0.0

        else:
            # Unknown or invalid state transition
            if "measure current entropy" in cmd:
```

```
                observation = (
                    f"You attempt to measure entropy. The system is in an unexpected state.\n\n"
                    f"Current cumulative entropy: {self.entropy_current:.4e} J/K\n\n"
                    f"Possible issue: process sequence may be incomplete or out of order.\n"
                    f"Ensure you follow the correct thermodynamic sequence: heating -> expansion -> cooling."
                )
                reward = 0.2
            elif "scan system parameters" in cmd:
                observation = (
                    f"System diagnostics available. Current state:\n"
                    f"  Step: {self.current_step}\n"
                    f"  Heating complete: {self.heating_complete}\n"
                    f"  Expansion complete: {self.expansion_complete}\n"
                    f"  Cooling complete: {self.cooling_complete}\n\n"
                    f"Entropy so far: {self.entropy_current:.4e} J/K"
                )
                reward = 0.3
            else:
                observation = (
                    f"Unexpected state interaction. The system is either in transition or already concluded.\n\n"
                    f"Available safe actions:\n"
                    f"  - measure current entropy\n"
                    f"  - scan system parameters"
                )
                reward = 0.0

        # Final check for termination via max turns
        if truncated and not self.goal_achieved:
            terminated = True
            reward = 0.7 if self.cooling_complete else 0.0  # partial credit only for a completed cycle
            observation = (
                f"Time limit reached ({self.max_turns} turns). The thermodynamic cycle was not fully verified.\n\n"
                f"Final entropy change: {self.entropy_current:.4e} J/K\n\n"
                f"The net entropy change was not confirmed as zero before the time limit.\n"
                f"Consider a more deliberate sequence in future attempts."
            )

        return observation, reward, terminated, truncated, info

    def solution(self) -> str:
        # Reference sequence that solves the environment
        return "activate heating, initiate expansion, begin cooling, measure current entropy"
```

G.3.3 Full Source of a Generated Tool-Use Environment

The late tool-use exemplar of the cards above in full: `CustomerSupportTicketWorkflowEnv`, generated at training step 392 of the 30B tool-use run. The environment subclasses the training harness's tool-use base class, which dispatches OpenAI-format tool calls to the `tool_*` methods, replays each instruction's criterion after every call, and terminates when the final answer is submitted after all criteria pass.

Training step 392: `CustomerSupportTicketWorkflowEnv` (200 lines).

```
"""SPADE self-play generated game"""

import random
import json
from typing import Tuple

class CustomerSupportTicketWorkflowEnv(ToolUseBaseEnv):
    def reset(self, seed=None) -> Tuple[str, dict]:
        self.turn_count = 0
        self._call_history = []
        if seed is not None:
            random.seed(seed)
        # Initialize CRM state with tickets, statuses, and assignments
        self._state = {
            'tickets': [
```

```
                {'id': 'TICKET-001', 'subject': 'Login Issues', 'status': 'new', 'priority': 'high', 'assigned_to': None, 'notes': []},
                {'id': 'TICKET-002', 'subject': 'Billing Discrepancy', 'status': 'in_progress', 'priority': 'medium', 'assigned_to': 'agent_01', 'notes': ['Contacted customer, awaiting response']},
                {'id': 'TICKET-003', 'subject': 'Feature Request: Dark Mode', 'status': 'new', 'priority': 'low', 'assigned_to': None, 'notes': []},
                {'id': 'TICKET-004', 'subject': 'Password Reset Failure', 'status': 'new', 'priority': 'high', 'assigned_to': None, 'notes': []},
                {'id': 'TICKET-005', 'subject': 'API Rate Limiting', 'status': 'in_progress', 'priority': 'high', 'assigned_to': 'agent_02', 'notes': ['Investigating backend logs']},
            ],
            'agents': {
                'agent_01': {'name': 'Sarah Chen', 'available': True, 'tickets_handled': 12},
                'agent_02': {'name': 'James Kim', 'available': True, 'tickets_handled': 8},
                'agent_03': {'name': 'Linda Wu', 'available': False, 'tickets_handled': 5}
            },
            'last_assigned_ticket': None,
            'last_updated_status': None,
            'last_added_note': None,
            'last_searched_ticket': None,
            'last_resolved_ticket': None,
            'last_reassigned_ticket': None,
            'last_action_success': False
        }
        self._tools = {
            'search_tickets': {
                'description': 'Search for tickets by subject keyword or status.',
                'parameters': {
                    'type': 'object',
                    'properties': {
                        'query': {'type': 'string', 'description': 'Keyword to search for in ticket subject or status.'}
                    },
                    'required': ['query']
                }
            },
            'update_ticket_status': {
                'description': 'Update the status of a ticket by its ID.',
                'parameters': {
                    'type': 'object',
                    'properties': {
                        'ticket_id': {'type': 'string', 'description': 'The ID of the ticket to update.'},
                        'status': {
                            'type': 'string',
                            'enum': ['new', 'in_progress', 'resolved', 'closed'],
                            'description': 'The new status for the ticket.'
                        }
                    },
                    'required': ['ticket_id', 'status']
                }
            },
            'assign_ticket': {
                'description': 'Assign a ticket to an available agent by agent ID.',
                'parameters': {
                    'type': 'object',
                    'properties': {
                        'ticket_id': {'type': 'string', 'description': 'The ID of the ticket to assign.'},
                        'agent_id': {'type': 'string', 'description': 'The ID of the agent to assign the ticket to.'}
                    },
                    'required': ['ticket_id', 'agent_id']
                }
            },
            'add_note_to_ticket': {
                'description': 'Add a note to a ticket for internal tracking.',
                'parameters': {
                    'type': 'object',
                    'properties': {
                        'ticket_id': {'type': 'string', 'description': 'The ID of the ticket to update.'},
                        'note': {'type': 'string', 'description': 'The note content to add.'}
                    },
                    'required': ['ticket_id', 'note']
                }
            },
            'resolve_ticket': {
```

```
                'description': 'Mark a ticket as resolved after confirming issue is fixed.',
                'parameters': {
                    'type': 'object',
                    'properties': {
                        'ticket_id': {'type': 'string', 'description': 'The ID of the ticket to resolve.'}
                    },
                    'required': ['ticket_id']
                }
            },
            'list_assigned_tickets': {
                'description': 'List all tickets currently assigned to a specific agent.',
                'parameters': {
                    'type': 'object',
                    'properties': {
                        'agent_id': {'type': 'string', 'description': 'The ID of the agent to list tickets for.'}
                    },
                    'required': ['agent_id']
                }
            }
        }
        self._user_messages = [
            "Find all high-priority tickets that are still in 'new' status.",
            "Assign the ticket with ID TICKET-001 to agent_01.",
            "Add a note to ticket TICKET-001 stating 'Customer confirmed issue is reproducible.'",
            "Update the status of ticket TICKET-001 to 'in_progress'.",
            "Finally, confirm that ticket TICKET-001 has been successfully resolved by marking it as 'resolved'."
        ]
        self._message_criteria = [
            lambda s: len([t for t in s['tickets'] if t['priority'] == 'high' and t['status'] == 'new']) == 2,
            lambda s: any(t['id'] == 'TICKET-001' and t['assigned_to'] == 'agent_01' for t in s['tickets']),
            lambda s: any(t['id'] == 'TICKET-001' and any('Customer confirmed issue is reproducible' in n for n in t['notes']) for t in s['tickets']),
            lambda s: any(t['id'] == 'TICKET-001' and t['status'] == 'in_progress' for t in s['tickets']),
            lambda s: any(t['id'] == 'TICKET-001' and t['status'] == 'resolved' for t in s['tickets'])
        ]
        self._current_msg = 0
        self._expected_answer = "done"
        return (self._user_messages[0], {})

    def _advance_if_done(self) -> str:
        if self._current_msg >= len(self._user_messages):
            return ""
        criterion = self._message_criteria[self._current_msg]
        if criterion(self._state):
            self._current_msg += 1
            if self._current_msg >= len(self._user_messages):
                return "\n\n[ALL STEPS COMPLETE] Submit <answer>done</answer>."
            next_msg = self._user_messages[self._current_msg]
            return f"\n\n[STEP {self._current_msg} COMPLETE -- NEW INSTRUCTION] {next_msg}"
        return ""

    def _check_answer(self, answer: str) -> bool:
        return answer.strip().lower() == 'done' and self._current_msg >= len(self._user_messages)

    def tool_search_tickets(self, query='') -> str:
        results = []
        for ticket in self._state['tickets']:
            if query.lower() in ticket['subject'].lower() or query.lower() in ticket['status']:
                results.append(ticket['id'])
        self._state['last_searched_ticket'] = query
        result_str = json.dumps(results)
        return result_str + self._advance_if_done()

    def tool_update_ticket_status(self, ticket_id='', status='') -> str:
        for ticket in self._state['tickets']:
            if ticket['id'] == ticket_id:
                ticket['status'] = status
                self._state['last_updated_status'] = ticket_id
                self._state['last_action_success'] = True
                return f"Updated ticket {ticket_id} status to {status}" + self._advance_if_done()
```

```
        return f"Error: Ticket {ticket_id} not found" + self._advance_if_done()

    def tool_assign_ticket(self, ticket_id='', agent_id='') -> str:
        # Check if agent exists and is available
        if agent_id not in self._state['agents']:
            return f"Error: Agent {agent_id} not found" + self._advance_if_done()
        if not self._state['agents'][agent_id]['available']:
            return f"Error: Agent {agent_id} is not available" + self._advance_if_done()

        for ticket in self._state['tickets']:
            if ticket['id'] == ticket_id:
                ticket['assigned_to'] = agent_id
                self._state['last_assigned_ticket'] = ticket_id
                self._state['last_action_success'] = True
                return f"Assigned ticket {ticket_id} to agent {agent_id}" +
                ↪  self._advance_if_done()
        return f"Error: Ticket {ticket_id} not found" + self._advance_if_done()

    def tool_add_note_to_ticket(self, ticket_id='', note='') -> str:
        for ticket in self._state['tickets']:
            if ticket['id'] == ticket_id:
                ticket['notes'].append(note)
                self._state['last_added_note'] = ticket_id
                self._state['last_action_success'] = True
                return f"Added note to ticket {ticket_id}: {note}" +
                ↪  self._advance_if_done()
        return f"Error: Ticket {ticket_id} not found" + self._advance_if_done()

    def tool_resolve_ticket(self, ticket_id='') -> str:
        for ticket in self._state['tickets']:
            if ticket['id'] == ticket_id:
                ticket['status'] = 'resolved'
                self._state['last_resolved_ticket'] = ticket_id
                self._state['last_action_success'] = True
                return f"Ticket {ticket_id} marked as resolved" + self._advance_if_done()
        return f"Error: Ticket {ticket_id} not found" + self._advance_if_done()

    def tool_list_assigned_tickets(self, agent_id='') -> str:
        assigned = [t['id'] for t in self._state['tickets'] if t['assigned_to'] ==
        ↪  agent_id]
        self._state['last_action_success'] = True
        return json.dumps(assigned) + self._advance_if_done()

    def solution(self) -> str:
        return (
            "1. tool_search_tickets(query='high priority new') "
            "2. tool_assign_ticket(ticket_id='TICKET-001', agent_id='agent_01') "
            "3. tool_add_note_to_ticket(ticket_id='TICKET-001', note='Customer confirmed
            ↪  issue is reproducible.') "
            "4. tool_update_ticket_status(ticket_id='TICKET-001', status='in_progress') "
            "5. tool_resolve_ticket(ticket_id='TICKET-001') "
            "6. <answer>done</answer>"
        )
```

H System Prompts and Templates

Environment-generation prompt. The `Environment Designer` receives the prompt below to generate a new single-turn game as executable Python; the skill name, its description, and a short list of example concepts fill the slots shown in angle brackets. In the canonical corpus-grounded runs the `Environment Designer` instead conditions on a freshly sampled corpus document, and the example-concept slot is unused. The single-turn prompt serves the single-turn setting of Section 3 (one derivation task, graded per attempt), while the multi-turn prompts demand stateful interaction. The `\boxed{}` extraction in the displayed contracts stops at the first closing brace, so generated environments use plain, brace-free answer strings.

Single-turn environment-generation prompt (`Environment Designer`)

```
<|im_start|>system
You are an expert Python programmer and game designer. You create
    educational language games for training Large Language Models.<|
    im_end|>
<|im_start|>user
Create a challenging single-player text-based game as a Python class
    that tests: <SKILL_NAME> (<SKILL_DESCRIPTION>).

GAME CONCEPT IDEAS (pick one or invent your own):
  <EXAMPLE_GAMES>

RULES:
- Be creative with the game concept - it can be a puzzle, riddle,
    logic problem, code tracing, word game, optimization task, or any
    reasoning challenge.
- One task per episode. Player answers with \boxed{answer}.
- Give your class a descriptive, unique name.
- The task must be DETERMINISTICALLY SOLVABLE from the observation
    alone - all information needed to derive the answer must be
    explicitly stated. No hidden state the player cannot see.
- The observation should be clearly written so a careful reader can
    follow the logic to the answer.

INFORMATION HIDING (critical for RL training):
- The observation must NOT contain the answer. The player must DERIVE
    it.
- For sequence/pattern games: show only partial data, never the target
     value.
- Never state the pattern rule directly - the player must discover it.
- Wrong-answer feedback must give hints (e.g., higher/lower, which
    part is wrong), never reveal the full solution.
- The player must use \boxed{answer} format - remind them in the
    observation.

DIFFICULTY:
- The task MUST require at least 3 distinct reasoning steps - not
    solvable in one or two arithmetic operations.
- Random guessing should succeed < 1% of the time - use large answer
    spaces.
- Use randomized parameters large enough that the answer is not
    obvious.
- Do NOT generate simple formula-substitution or single-operation
    problems.
- Compute the solution from the generated puzzle; never hardcode it.
  WRONG: return 'recursive'  # same answer regardless of puzzle
  WRONG: return str(random.randint(1,10))  # not derived from the
    actual task
  RIGHT: generate puzzle first, then compute self._solution from it
```

````
INTERFACE CONTRACT:
```python
import random
import re
from typing import Any, Optional, Tuple, Dict, List
from math_verify import parse, verify

def verify_answer(player_answer, solution):
    """Check if answer is correct: string match first, then math
    equivalence."""
    if str(player_answer).strip() == str(solution).strip():
        return True
    try:
        return verify(parse(player_answer), parse(solution))
    except:
        return False

class DescriptiveNameEnv:
    def __init__(self, max_turns=10, **kwargs):
        ...
        self.reset()

    def reset(self, seed=None) -> Tuple[str, dict]:
        # Generate a new task. Returns (observation_with_instructions,
     {})
        ...

    def solution(self) -> str:
        # REQUIRED: Return the exact answer string that solves the
    current task.
        ...

    def step(self, action: str) -> Tuple[str, float, bool, bool, dict
    ]:
        self.turn_count += 1
        truncated = self.turn_count >= self.max_turns
        match = re.search(r'\\boxed\{([^}]+)\}', action)
        if match and verify_answer(match.group(1), self.solution()):
            return ("Correct! ...", 1.0, True, False, {})
        elif truncated:
            return (f"Time's up. The answer was {self.solution()}. [
    task]", 0.0, False, True, {})
        else:
            return ("Wrong. [specific hint about why]. Try again. [
    task description]", 0.0, False, False, {})
        # EVERY code path must return (str, float, bool, bool, dict) -
     no exceptions.

    def close(self): pass
```

EPISODE STRUCTURE (critical for RL training):
1. reset(): Generate ONE task/puzzle for this episode
2. step(): Player tries to solve that SAME task multiple times
3. Correct answer -> terminated=True (episode ends with success)
4. Max turns reached -> truncated=True, reveal the solution so the
    model learns from failure
5. NEVER generate a new task inside step() - the task is fixed for the
     whole episode

OBSERVATION REQUIREMENTS:
- Turn 1 (reset): Show instructions + task + remind player to use \
    boxed{answer} format
````

```
- Turn 2+ (step): Show specific feedback on the attempt + the SAME
    task (no instructions)
- The task/puzzle MUST be visible every turn or the model won't know
    what to solve
- Feedback must be specific to the attempted answer, not generic ('
    Wrong. Try again.' is not enough)
- No \boxed{} in action: remind the player of the format + show task (
    do not terminate)

ROBUSTNESS:
- Never divide by a value that could be zero - regenerate if needed
- Initialize ALL instance variables in __init__ before calling reset()
- Return empty dict {} for info (never strings)
- Every code path in step() must return a 5-tuple (str, float, bool,
    bool, dict).
- Naming: never store data in self.solution - that shadows the
    required solution() method. Use self._solution or self._answer
    instead.
- Brace escaping: in f-strings, {word} evaluates word as a Python
    expression. To write a literal \boxed{answer} escape it as \boxed
    {{answer}}. Same rule with .format(): any literal brace must be
    doubled.

Generate the complete Python code in a ```python block.<|im_end|>
<|im_start|>assistant
```

Multi-turn environment-generation prompt. Because gameplay spans multiple turns, the `Environment Designer` also receives a dedicated prompt for generating interactive, multi-turn games (state that evolves across turns, branching actions), shown below with the same placeholder convention.

Multi-turn environment-generation prompt (`Environment Designer`)

```
<|im_start|>system
You are an expert Python programmer and game designer specializing in
    interactive, multi-turn text-based games. You create environments
    where the player must make sequential decisions across multiple
    turns, with each action changing the game state.<|im_end|>
<|im_start|>user
Create an interactive multi-turn text-based game as a Python class
    that tests: <SKILL_NAME> (<SKILL_DESCRIPTION>).

GAME CONCEPT IDEAS for <SKILL_NAME> (pick one or invent similar):
  - <EXAMPLE_GAMES>

WHAT MAKES A GOOD MULTI-TURN GAME:
Think of classic text adventures, board games, or strategy games. The
    player
navigates a world that changes with every action. Good examples:
  - Grid exploration: move through rooms, find keys to unlock doors,
    reach the exit
  - Trading: buy low / sell high across rounds with fluctuating prices
  - Survival: manage health, food, tools while exploring - wrong
    choices kill you
  - Investigation: question suspects, search locations, piece together
     clues
  - Tower defense: place defenses, then waves arrive - adapt strategy
    each wave
  - Crafting: gather materials, combine them in the right order to
    build something
```

```
WHAT TO AVOID (common failure modes):
  BAD: A math puzzle where the player guesses the answer and retries
    on failure.
       That is a single-turn puzzle with retry, NOT a multi-turn game.
  BAD: Increment/decrement a variable until it matches a target.
       That is a linear search, not a game - there are no decisions.
  BAD: The player has only one reasonable action each turn (e.g.,
    always 'allocate').
       If the optimal path is obvious, there is no strategic depth.
  BAD: The game can be solved in 1-3 turns. Too short for multi-turn
    training.
  BAD: Embedding single-turn math/logic puzzles inside a multi-turn
    frame.
       E.g., 'go to forest, solve 2+3, collect wood' - the actual
    decisions are trivial.
  BAD: Using input() to get player answers. NEVER use input(). The
    action parameter
       to step() IS the player's full response. Parse everything from
    it.

DESIGN REQUIREMENTS:
- The game world has STATE that changes on every action (positions,
    inventories,
  health, money, unlocked areas, NPC attitudes, etc.).
- Each turn, the player chooses from 2+ meaningfully different actions
    .
- Some actions are BETTER than others - wrong choices waste turns or
    cause harm.
- The game will be played for at most 20 turns. Design difficulty,
  resource budgets, and pacing accordingly. Optimal play should win in
  roughly 10-15 turns.
- Random play should lose most of the time.
- The game must have a clear WIN condition and ideally a LOSE
    condition too.

ACTION FORMAT:
- The player wraps every action in \boxed{action}. The step() method
    extracts
  the content inside \boxed{} and interprets it.
- Show available actions clearly in every observation, e.g.:
    Actions: \boxed{go north}, \boxed{go south}, \boxed{pick up key},
    \boxed{rest}
- If the player's input has no \boxed{}, return a format reminder. Do
    NOT terminate.

OBSERVATION FORMAT:
- Each observation must show: (1) current state, (2) result of last
    action,
  (3) available actions, (4) progress (e.g., 'Turn 3/12 | HP: 7/10 |
    Items: [key]').
- The initial observation (from reset) includes rules, goal, and
    starting state.

REWARD:
- Win: reward = 1.0, terminated = True
- Lose: reward = 0.0, terminated = True
- Intermediate turns: reward = 0.0
- Max turns without winning: reward = 0.0, truncated = True

INTERFACE:
```python
import random
import re
```
```

````
from typing import Tuple
from math_verify import parse, verify

def verify_answer(player_answer, solution):
    """Check if answer is correct: string match first, then math
    equivalence."""
    if str(player_answer).strip() == str(solution).strip():
        return True
    try:
        return verify(parse(player_answer), parse(solution))
    except:
        return False

class DescriptiveNameEnv:
    def __init__(self, max_turns=20, **kwargs):
        self.max_turns = max_turns
        self.turn_count = 0
        # ALL state variables initialized here
        self.reset()

    def reset(self, seed=None) -> Tuple[str, dict]:
        if seed is not None:
            random.seed(seed)
        self.turn_count = 0
        # Randomize the game world
        return observation, {}

    def step(self, action: str) -> Tuple[str, float, bool, bool, dict
    ]:
        self.turn_count += 1
        match = re.search(r'\\boxed\{([^}]*)\}', action)
        if not match:
            return ('Use \\boxed{action} format.', 0.0, False, False,
    {})
        cmd = match.group(1).strip().lower()
        # Update state, check win/lose
        # Use verify_answer(cmd, solution) for numeric comparisons
        truncated = self.turn_count >= self.max_turns
        return (observation, reward, terminated, truncated, {})

    def close(self): pass
```

REFERENCE EXAMPLE - Trading Game (shows the pattern; yours must be
    DIFFERENT):
```python
class TradingGameEnv:
    """Buy low, sell high across rounds with fluctuating prices. Reach
     target gold."""
    def __init__(self, max_turns=12, **kwargs):
        self.max_turns = max_turns
        self.turn_count = 0
        self.gold = 0
        self.stock = 0
        self.price = 0
        self.price_history = []
        self.target_gold = 0
        self.trend = 0  # hidden: +1 rising, -1 falling
        self.reset()

    def reset(self, seed=None):
        if seed is not None:
            random.seed(seed)
````

```
        self.turn_count = 0
        self.gold = 100
        self.stock = 0
        self.price = random.randint(8, 15)
        self.price_history = [self.price]
        self.target_gold = 200
        self.trend = random.choice([-1, 1])
        return (f'Welcome to the Trading Game! Reach {self.target_gold
    } gold to win.\n'
                f'Price: {self.price} | Gold: {self.gold} | Stock: {
    self.stock}\n'
                f'History: {self.price_history}\n'
                f'Turn 0/{self.max_turns}\n'
                f'Actions: \\boxed{{buy N}}, \\boxed{{sell N}}, \\
    boxed{{wait}}'), {}

    def step(self, action):
        self.turn_count += 1
        match = re.search(r'\\boxed\{([^}]*)\}', action)
        if not match:
            return ('Use \\boxed{action} format.', 0.0, False, False,
    {})
        cmd = match.group(1).strip().lower()
        truncated = self.turn_count >= self.max_turns
        msg = ''
        # Parse action
        if cmd.startswith('buy '):
            try:
                n = int(cmd[4:])
                cost = n * self.price
                if cost <= self.gold and n > 0:
                    self.gold -= cost
                    self.stock += n
                    msg = f'Bought {n} at {self.price}.'
                else:
                    msg = f'Cannot buy {n} (need {cost} gold, have {
    self.gold}).'
            except ValueError:
                msg = 'Invalid number.'
        elif cmd.startswith('sell '):
            try:
                n = int(cmd[5:])
                if n <= self.stock and n > 0:
                    self.gold += n * self.price
                    self.stock -= n
                    msg = f'Sold {n} at {self.price}.'
                else:
                    msg = f'Cannot sell {n} (have {self.stock}).'
            except ValueError:
                msg = 'Invalid number.'
        elif cmd == 'wait':
            msg = 'You wait.'
        else:
            msg = 'Unknown action.'
        # Update price with trend + noise
        if random.random() < 0.3:  # trend reversal
            self.trend *= -1
        self.price = max(1, self.price + self.trend * random.randint
    (1, 4))
        self.price_history.append(self.price)
        # Check win
        total = self.gold + self.stock * self.price
        if self.gold >= self.target_gold:
```

```
            return (f'{msg} Gold: {self.gold}. You win!', 1.0, True,
    False, {})
        if truncated:
            return (f'{msg} Time up. Gold: {self.gold}, Stock: {self.
    stock}. Needed {self.target_gold}.', 0.0, False, True, {})
        obs = (f'{msg}\nPrice: {self.price} | Gold: {self.gold} |
    Stock: {self.stock}\n'
               f'History: {self.price_history[-5:]}\n'
               f'Turn {self.turn_count}/{self.max_turns}\n'
               f'Actions: \\boxed{{buy N}}, \\boxed{{sell N}}, \\boxed
    {{wait}}')
        return (obs, 0.0, False, False, {})

    def close(self): pass
```

ROBUSTNESS:
- NEVER use input(). The step(action) parameter IS the player's response.
- Never divide by a value that could be zero.
- Initialize ALL instance variables in __init__ before calling reset().
- Return empty dict {} for info (never strings).
- Every code path in step() must return a 5-tuple (str, float, bool, bool, dict).
- Handle unexpected player input gracefully (show valid actions, don't crash).
- Use only the Python standard library (random, re, collections, etc.).
- Brace escaping: in f-strings, literal braces must be doubled {{ }}.
- NEVER put backslashes inside f-string expressions. Use a variable instead:
  BAD: f'{"\n".join(items)}'
  GOOD: sep = '\n'; f'{sep.join(items)}' OR '\n'.join(items)

Generate the complete Python code in a ```python block.
Remember: the game must have BRANCHING DECISIONS where different choices
lead to different outcomes. A game with only one reasonable action per turn is not acceptable.<|im_end|>
<|im_start|>assistant

Corpus-grounded interactive generation prompt. The canonical games runs use the corpus-grounded variant below: the sampled document replaces the example-concept list, and the specification demands hidden state, multi-turn structure, and partial reward. The skill, difficulty, and document slots are shown in angle brackets.

Corpus-grounded multi-turn environment-generation prompt (`Environment Designer`)

You are given a reference document. Create an INTERACTIVE, MULTI-TURN Python game
environment grounded in a concept/technique from this document.

<REFERENCE_DOCUMENT>
<DOCUMENT_TEXT>
</REFERENCE_DOCUMENT>
This MUST be a genuine multi-turn INTERACTIVE environment, NOT a one-shot question-and-answer quiz.

HOW THE AGENT ACTS (this matches the training pipeline - follow it exactly):

```
- Every turn the agent submits ONE action wrapped as \boxed{<action>},
    e.g. \boxed{measure node 3},
  \boxed{move north}, \boxed{open valve A}. step() receives that
    string; extract the action with
  re.search(r'\\boxed\{(.+?)\}', action) and treat the extracted text
    as an interactive COMMAND.
- The extracted text is a COMMAND / move / query that CHANGES the
    environment - it is NOT a final
  answer to grade once. step() must UPDATE the environment's state
    from it and return a NEW
  observation reflecting the changed state.

WHAT MAKES IT MULTI-TURN (required):
- The goal CANNOT be reached in one action. It needs a SEQUENCE of
    actions - explore to uncover
  hidden information, then act on it; or manipulate state step-by-step
     toward a target.
- HIDDEN STATE: the agent does NOT see everything at reset(); it must
    act to reveal/probe state
  across turns (the observation grows/changes as it acts).
- PARTIAL reward for progress toward the goal; terminated=True only
    when the goal is reached.
- Each turn's observation must (a) reflect the updated state and (b)
    remind the agent to reply
  with its next action as \boxed{<action>}.

DO NOT (these collapse it back to single-shot QA):
- Do NOT extract \boxed{answer}, grade it once, and terminate. \boxed
    carries a per-turn COMMAND,
  not a final answer.
- Do NOT state the full problem at reset() and just check one
    submitted value.
- Do NOT repeat the SAME static task every turn with only 'wrong, try
    again' feedback.

GROUNDING: the interaction must require understanding a concept/
    technique from the document
(e.g. if it describes an algorithm, the agent EXECUTES it step-by-step
     interactively; if it
describes a system, the agent OPERATES it over turns). Never reference
     the document in the game
text (no 'according to the passage'); write it as a standalone
    environment.

INTERFACE CONTRACT (interactive; \boxed wraps each turn's COMMAND):
```python
import random, re
from typing import Tuple, Dict

class DescriptiveInteractiveEnv:
    def __init__(self, max_turns=12, **kwargs):
        self.reset()
    def reset(self, seed=None) -> Tuple[str, dict]:
        self.turn_count = 0
        # set up HIDDEN state + a goal needing several actions;
    randomize by seed.
        # return (observation: situation + AVAILABLE ACTIONS + 'reply
    with \boxed{<action>}',
        #         but NOT the full solution), {}
        ...
    def step(self, action: str) -> Tuple[str, float, bool, bool, dict
    ]:
        self.turn_count += 1
```
```

```
        truncated = self.turn_count >= self.max_turns
        m = re.search(r'\\boxed\{(.+?)\}', action)
        cmd = (m.group(1) if m else action).strip()      # the per-
    turn COMMAND
        # 1) PARSE cmd into an operation; 2) UPDATE self state; 3)
    build a NEW observation
        #    reflecting the updated state; 4) reward = partial
    progress in [0,1],
        #    terminated=True only when the goal is reached.
        # every code path returns (str, float, bool, bool, dict)
        ...
    def solution(self) -> str:
        # a reference action-sequence (or goal description) that
    solves it
        ...
    def close(self): pass
```

```
REWARD SCALE: keep total reward in [0, 1]; partial progress < 1.0,
    full success = 1.0.

ROBUSTNESS (the most common failures - follow ALL of these or the env
    won't load):
- BRACE ESCAPING: in f-strings and .format(), {word} evaluates word;
    to write a LITERAL brace you
  must DOUBLE it. To print \boxed{<action>} inside an f-string, write
    \boxed{{<action>}}.
- Initialize ALL instance variables in __init__ BEFORE calling reset()
     (no AttributeError mid-episode).
- Imports: use ONLY the Python standard library plus `math` and `
    random`. Import everything you use;
  never call an unimported name (no bare `integrate`, no `np.`, no `x.
    cos()` - use `math.cos(x)`).
- Every step() code path returns the 5-tuple (str, float, bool, bool,
    dict); info is always a dict {}.
- Never crash on an unrecognized/garbage command - return a helpful
    observation + reward 0.0 instead.
- Never divide by a value that could be zero; never store data in self
    .solution (use self._solution).

SELF-VERIFICATION before finalizing:
- Trace TWO different action sequences from reset(seed=0): does the
    observation CHANGE based on the
  actions? (If it never changes, it is NOT interactive - fix it.)
- Does \boxed carry a per-turn COMMAND that evolves state, rather than
     a final answer that ends it?
- Is partial reward given for progress, kept in [0,1]?

Target skill: <SKILL_NAME> (<SKILL_DESCRIPTION>). DIFFICULTY: <
    DIFFICULTY> - the goal
should need <N>+ interaction steps.

Generate the complete Python code in a ```python block.
```

Tool-use generation prompt. The tool-use runs generate environments with the multi-turn variant below (selected by `SPADE_MULTITURN_ENV_GEN=1` in the released launchers). The skill, difficulty, and example slots fill as in the games prompts; each generation call additionally samples one of ten application domains and one of four task variants (weighted 3:2:1:1), whose filler blocks follow the main template.

Multi-turn tool-use environment-generation prompt (`Environment Designer`)

```
Create a MULTI-TURN tool-use environment by subclassing ToolUseBaseEnv
    .
The environment tests: __SKILL_NAME__ (__SKILL_DESCRIPTION__).

THIS TASK MUST MIRROR THE STRUCTURE OF A REAL MULTI-TURN INTERACTION.
A real user does NOT describe the entire workflow up front. Instead,
    they
issue ONE atomic instruction, wait for it to be done, then issue the
    next.
Your env must reproduce this turn-by-turn structure.

----------------------------------------------------------------
WHAT YOU IMPLEMENT (the base class handles parsing/dispatch):

  - reset(seed) -- generate state, define self._tools, define
       self._user_messages (list of 3-5 atomic instruction strings),
       self._message_criteria (list of callables: state -> bool),
       self._current_msg = 0, self._expected_answer = "done".
       Return (self._user_messages[0], {})

  - tool_xxx(**kwargs) -> str -- one method per tool. After mutating
       state, EACH tool method must call self._advance_if_done() and
       APPEND its output to the tool result, like:
         base_result = "Moved file.pdf to /temp"
         progress = self._advance_if_done()  # appends next
    instruction or completion marker
         return base_result + progress

  - solution() -> str -- describe the full multi-turn solution

  - _advance_if_done(self) -> str -- helper you write yourself:
    ```
      def _advance_if_done(self):
          if self._current_msg >= len(self._user_messages):
              return ""
          criterion = self._message_criteria[self._current_msg]
          if criterion(self._state):
              self._current_msg += 1
              if self._current_msg >= len(self._user_messages):
                  return "\n\n[ALL STEPS COMPLETE] Submit <answer>done
    </answer>."
              next_msg = self._user_messages[self._current_msg]
              return f"\n\n[STEP {self._current_msg} COMPLETE - NEW
    INSTRUCTION] {next_msg}"
          return ""
    ```

  - Override _check_answer(answer) so it returns True only if
       answer == "done" AND self._current_msg == len(self.
    _user_messages).

----------------------------------------------------------------
USER MESSAGE STRUCTURE (THE CORE OF MULTI-TURN):

self._user_messages must be 3-5 ATOMIC instructions, e.g.:
  [
    "Navigate to the document folder.",
    "Move final_report.pdf to a new 'temp' subdirectory.",
    "Now create an 'archive' folder and move all .txt files there.",
    "Finally, list the contents of the archive folder.",
  ]
```

```
self._message_criteria are callables that read self._state and return
    True
when the criterion is met:
  [
    lambda s: s["cwd"] == "/document",
    lambda s: "final_report.pdf" in s["tree"].get("/document/temp",
    []),
    lambda s: all(f in s["tree"].get("/document/archive", []) for f in
     s["txt_files"]),
    lambda s: s["last_ls_path"] == "/document/archive",
  ]

----------------------------------------------------------------
DOMAIN RULES:

- Each user message is a SHORT, SPECIFIC, ATOMIC instruction (1-2
    sentences).
- Do NOT describe the whole workflow in advance. The user reveals
  instructions ONE AT A TIME as previous ones are completed.
- Do NOT mention tool names, file structures, or implementation
    details
  in user messages. Use natural-language goals only.
- Each instruction's success criterion must be checkable from self.
    _state.
- The actor must complete the CURRENT message before progressing.
    Trying
  to skip ahead won't reveal future messages.
- Allow 4-8 tool calls per message.

----------------------------------------------------------------
CRITICAL - SOLVABILITY & CRITERION CORRECTNESS (the single biggest
    source of BROKEN games):

A game is BROKEN if a success criterion cannot be satisfied by
    following its
instruction, or is already satisfied before the agent acts. Broken
    games waste
training and produce deadlocks. Obey ALL of these:

1. FALSE AT RESET / NOT FREE. Every criterion MUST require the agent
    to actually
   perform its instruction's action. For a WRITE instruction, the
    criterion must
   check the CHANGE it makes and be False on the state reset() returns
     - a
   criterion already True at reset (e.g. `s['inventory']['monitor'] >=
     5` when
   monitor starts at 8, or `any('phone' in i for i in s['electronics
    '])` when a
   phone is already stocked) lets the agent skip the step for free.
    For a READ /
   VERIFY instruction ("check the line is active", "confirm the
    balance"), do NOT
   use a criterion that is already True at reset (e.g. `s['line_status
    ']=='active'`
   when it starts 'active') - that advances on ANY tool call without
    the agent
   doing the read. Instead have the read tool record that it ran (e.g.
     set
   `s['last_checked']='line_status'`) and make the criterion check
    THAT flag, so
   the step requires actually calling the read tool.
```

```
2. TYPE-CORRECT. The criterion must read self._state with the SAME
    shape the
   tools write it. If order_history is a list of DICTS, check
   `any(o['item']=='laptop' for o in s['order_history'])`, NOT
   `'laptop' in s['order_history']` - string-in-list-of-dicts is
    ALWAYS False, so
   the step can NEVER complete and the game deadlocks. Type mismatches
     are the
   most common unsolvable-game bug.

3. DERIVABLE BY THE AGENT. Every value a criterion requires must be
    obtainable by
   the agent from EITHER (a) the words of its instruction, OR (b) a
    tool result it
   can read. NEVER hide an exact date / id / amount / threshold the
    instruction
   does not state and no tool reveals. BAD: instruction "schedule a
    visit next
   week" but criterion `s['last_updated'] >= '2024-01-22'` - the agent
     cannot know
   the threshold, must guess, and usually fails. FIX: state it in the
    instruction
   ("schedule for 2024-01-22 or later"), OR accept ANY valid action
   (`s['last_updated'] != '2024-01-15'`, i.e. any new date), OR have a
     tool reveal
   the value. Same for ids: if a criterion needs order '1024', the
    instruction
   must name it or a tool must return it.

4. EVERY REQUIRED TOOL CAN SUCCEED. For each instruction, the tool
    that satisfies
   its criterion must have a reachable success path. Trace the field
    the tool
   looks up - it must be a key some tool (or reset) actually SETS. BAD
    :
   tool_process_refund finds a refund by `r['refund_id']` but
    tool_create_refund
   never stores a 'refund_id' key -> process_refund can NEVER succeed.
     If a tool
   reads an id/field, an earlier tool or reset MUST write that exact
    key.

5. INSTRUCTION MATCHES CRITERION. Each instruction must describe
    exactly what its
   criterion checks - no more, no less. Do NOT write "after the refund
     is
   processed, list inventory" if the criterion only checks that
    inventory was
   listed (and process_refund is broken/unneeded): the agent chases
    the
   irrelevant step and stalls.

6. SELF-TRACE BEFORE YOU FINISH (REQUIRED). Mentally run reset(seed=0)
    , then
   execute your own solution() sequence step by step. After EACH
    solution step the
   matching criterion must flip False->True, IN ORDER, and no later
    criterion may be
   True yet; at the end self._current_msg must equal len(self.
    _user_messages) and
   _check_answer("done") must be True. If any criterion is True at
    reset, never
```

```
   becomes True, or throws (KeyError/IndexError), the game is BROKEN -
     fix it
   before returning.

------------------------------------------------------------------
DIFFICULTY: __DIFFICULTY__

__DOMAIN_BLOCK__
__TASK_TYPE_BLOCK__
SKILL FOCUS: __SKILL_CATEGORY__
EXAMPLE TASKS (pick one or invent your own):
  __EXAMPLES__

__CORPUS_SECTION__

------------------------------------------------------------------
RULES:
- Class MUST inherit from ToolUseBaseEnv.
- Do NOT import ToolUseBaseEnv. Just write: class MyEnv(ToolUseBaseEnv
    ):
- Only standard library imports (random, json, re, math). No custom
    packages.
- Tools simulated as methods. No real APIs, network, or subprocess.
- Tools must be DETERMINISTIC. All randomness in reset() only.
- 3-8 tools as tool_xxx() methods returning strings.
- Give at least one tool a typed/constrained parameter using 'enum'
    and 'required' in its
  schema (e.g. status: {'type':'string','enum':['pending','shipped','
    cancelled']}).
- Include 1-2 DISTRACTOR tools: plausible but wrong for the task, so
    the agent must select
  the correct one from alternatives.
- Each tool ENDS with `return base_result + self._advance_if_done()`.
- Mutation tools must error gracefully on bad preconditions
  (e.g., mv without target dir -> "Error: target dir not found").
- CRITICAL: tool method parameter names MUST EXACTLY MATCH the names
    declared
  in self._tools[name]['parameters']['properties']. If schema says
  {'account': {...}}, the method MUST be `def tool_check(self, account
    =...)`.
  NOT `account_type`, NOT `acct`, NOT a renamed alias. Mismatched
    names
  cause every tool call to fail with "got an unexpected keyword
    argument".

------------------------------------------------------------------
EXAMPLE STRUCTURE (multi-turn filesystem task):

```python
import random
import json
from typing import Tuple

class FileWorkflowMTEnv(ToolUseBaseEnv):
    def reset(self, seed=None) -> Tuple[str, dict]:
        self.turn_count = 0
        self._call_history = []
        if seed is not None:
            random.seed(seed)
        self._state = {
            'cwd': '/home',
            'tree': {
                '/home': ['document', 'temp.txt'],
```
```

```
                '/home/document': ['report.txt', 'notes.txt', 'log.txt
'],
            },
            'last_ls_path': None,
        }
        self._tools = {
            'pwd': {'description': 'Show current dir', 'parameters':
{'type':'object','properties':{}}},
            'ls': {'description': 'List a directory', 'parameters': {'
type':'object','properties':{'path':{'type':'string'}}, 'required
':['path']}},
            'cd': {'description': 'Change dir', 'parameters': {'type
':'object','properties':{'path':{'type':'string'}}, 'required':['
path']}},
            'mkdir': {'description': 'Create a subdirectory in current
 dir', 'parameters': {'type':'object','properties':{'name':{'type
':'string'}}, 'required':['name']}},
            'mv': {'description': 'Move file from current dir to a
destination dir', 'parameters': {'type':'object','properties':{'
src':{'type':'string'},'dst':{'type':'string'}}, 'required':['src
','dst']}},
        }
        self._user_messages = [
            "Navigate to the document folder.",
            "Create a 'temp' subdirectory and move report.txt into it
.",
            "Now list what's in the temp folder.",
        ]
        self._message_criteria = [
            lambda s: s['cwd'] == '/home/document',
            lambda s: 'report.txt' in s['tree'].get('/home/document/
temp', []),
            lambda s: s['last_ls_path'] == '/home/document/temp',
        ]
        self._current_msg = 0
        self._expected_answer = "done"
        return (self._user_messages[0], {})

    def _advance_if_done(self) -> str:
        if self._current_msg >= len(self._user_messages):
            return ""
        if self._message_criteria[self._current_msg](self._state):
            self._current_msg += 1
            if self._current_msg >= len(self._user_messages):
                return "\n\n[ALL STEPS COMPLETE] Submit <answer>done</
answer>."
            next_msg = self._user_messages[self._current_msg]
            return f"\n\n[STEP {self._current_msg} COMPLETE - NEW
INSTRUCTION] {next_msg}"
        return ""

    def _check_answer(self, answer: str) -> bool:
        return answer.strip().lower() == 'done' and self._current_msg
>= len(self._user_messages)

    def tool_pwd(self) -> str:
        return self._state['cwd'] + self._advance_if_done()

    def tool_ls(self, path='') -> str:
        if path not in self._state['tree']:
            return f"Error: {path} not found" + self._advance_if_done
()
        self._state['last_ls_path'] = path
```

```
        out = json.dumps(self._state['tree'][path])
        return out + self._advance_if_done()

    def tool_cd(self, path='') -> str:
        if path not in self._state['tree']:
            return f"Error: {path} not found" + self._advance_if_done
    ()
        self._state['cwd'] = path
        return f"Changed to {path}" + self._advance_if_done()

    def tool_mkdir(self, name='') -> str:
        new_path = f"{self._state['cwd']}/{name}"
        if new_path in self._state['tree']:
            return f"Error: {new_path} exists" + self._advance_if_done
    ()
        self._state['tree'][new_path] = []
        self._state['tree'][self._state['cwd']].append(name)
        return f"Created {new_path}" + self._advance_if_done()

    def tool_mv(self, src='', dst='') -> str:
        cwd_files = self._state['tree'].get(self._state['cwd'], [])
        if src not in cwd_files:
            return f"Error: {src} not in current dir" + self.
    _advance_if_done()
        dst_path = f"{self._state['cwd']}/{dst}"
        if dst_path not in self._state['tree']:
            return f"Error: target {dst} not found" + self.
    _advance_if_done()
        cwd_files.remove(src)
        self._state['tree'][dst_path].append(src)
        return f"Moved {src} to {dst}" + self._advance_if_done()

    def solution(self) -> str:
        return ("1. cd(path='/home/document') 2. mkdir(name='temp') "
                "3. mv(src='report.txt', dst='temp') 4. ls(path='/home
    /document/temp') "
                "5. <answer>done</answer>")
```

Notice:
- self._user_messages defines the multi-turn STRUCTURE
- Each tool_xxx() ends with self._advance_if_done() so when criterion is met,
  the actor sees "[STEP N COMPLETE - NEW INSTRUCTION] ..." inline in the result
- _check_answer requires "done" AND all messages completed
- Reset returns ONLY the first instruction; the rest are revealed progressively

Generate the complete Python code in a ```python block.

Domain and task-variant filler blocks

__DOMAIN_BLOCK__ (one of ten application domains per call):

REAL-WORLD DOMAIN (build the tools, state, and user goal for THIS domain; do NOT
use the reference snippet's academic subject as the domain):
  __DOMAIN__
The task must be a realistic user accomplishing a goal here (search, book, order,

```
update, cancel, transfer, schedule) - NOT an exam or puzzle about any
    document.

__TASK_TYPE_BLOCK__ (one per call; execute and read_then_write are the
     common cases):

TASK VARIANT: execute (happy-path). The user issues atomic
    instructions; the
agent calls the right tools in order. Each criterion checks the
    resulting state.
TASK VARIANT: read_then_write. At least ONE instruction must force the
     agent to
FIRST query state with a read tool, branch on a predicate over the
    result (e.g.
'for every order over $100', 'all lines past their data cap'), THEN
    make the
dependent write. Its criterion must check the final state reflects the
predicate-conditioned writes - not merely that one tool was called.
TASK VARIANT: missing_value. At least ONE instruction references a
    value the user
does NOT state (e.g. 'cancel my most recent order' with no order_id; '
    pay the
overdue invoice' with no invoice id). The agent must DISCOVER it via a
     read tool
before the write. The write tool MUST return an error if given a wrong
    /guessed
value, so a fabricated argument cannot satisfy the criterion.
TASK VARIANT: with_irrelevant. Include EXACTLY ONE instruction that
    NONE of the
domain tools can satisfy (e.g. in a retail env, 'what will the weather
     be
tomorrow?'). Add a tool named 'decline_request' (one required string
    param
'reason') whose method sets a state flag and then advances; that
    instruction's
criterion is met ONLY by calling decline_request. Provide NO other
    tool that
could plausibly fulfill the irrelevant request.
```

`Reasoning Agent` gameplay prompt. The `Reasoning Agent` receives each turn's observation wrapped in the template below and answers in `\boxed{}` form.

`Reasoning Agent` gameplay prompt

```
<|im_start|>user
You are playing a language game. Make valid actions to win.
Observation: <OBSERVATION>
Please reason step by step, and put your final answer within \boxed
    {}.<|im_end|>
<|im_start|>assistant
```